PMD Resource Access Protocol
Programmer's Reference

P M D
Performance Motion Devices, Inc.
80 Central Street
Boxborough, MA 01719

Revision 1.1, April 18, 2012

NOTICE

his 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 PMD.

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 PMD.

Copyright 2009-2012 by Performance Motion Devices, Inc.

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

Warranty

PMD warrants performance of its products to the specifications applicable at the time of sale in accordance with PMD's standard warranty. Testing and other quality control techniques are utilized to the extent PMD deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Performance Motion Devices, Inc. (PMD) reserves the right to make changes to its products or to discontinue any product or service without notice, and advises customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

Safety Notice

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage. Products are not designed, authorized, or warranted to be suitable for use in life support devices or systems or other critical applications. Inclusion of PMD products in such applications is understood to be fully at the customer's risk.

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

PMD assumes no liability for applications assistance or customer product design. PMD 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 PMD covering or relating to any combination, machine, or process in which such products or services might be or are used. PMD's publication of information regarding any third party's products or services does not constitute PMD's approval, warranty or endorsement thereof.

Related Documents

Prodigy®/CME Stand-Alone Users Guide

Prodigy®/CME PCI Users Guide

Prodigy®/CME PC/104 Users Guide

Complete description of the Prodigy/CME motion cards including getting starting section, operational overview, detailed connector information, and complete electrical and mechanical specifications.

Magellan® Motion Processor User's Guide

Complete description of the Magellan Motion Processor features and functions with detailed theory of its operation.

Magellan® Motion Processor Programmer's Command Reference

Descriptions of all Magellan Motion Processor commands, with coding syntax and examples, listed alphabetically for quick reference.

C-Motion® Engine Development Tools

Description of the C-Motion Engine, development environment, and software tools.

Pro-Motion® User's 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.

ION Digital Drive User's Manual

Complete description of the ION Digital Drive including getting starting section, operational overview, detailed connector information, and complete electrical and mechanical specifications.

ION/CME Digital Drive User's Manual

Complete description of the ION/CME Digital Drive including getting starting section, operational overview, detailed connector information, and complete electrical and mechanical specifications.

ION/B Digital Drive User's Manual

Complete description of the ION/B Digital Drive including getting starting section, operational overview, detailed connector information, and complete electrical and mechanical specifications.

Table of Contents

Chapter 1. Introduction 7
1.1 PMD Resource Access Protocol Overview. 7
1.2 Scope 7

Chapter 2. PMD Resource Access Protocol (PRP) Tutorial 9
2.1 Resource Addressing 9
2.2 Accessing the Communications Ports 11
2.3 Accessing On-Card Resources. 15
2.4 Accessing Magellan-Attached Devices 16
2.5 PRP Communication Formats 18

Chapter 3. PRP Reference 21
3.1 PRP Resources. 21
3.2 PRP Addresses. 22
3.3 PRP Actions and Sub-Actions 22
3.4 PRP Packet Structure 23
3.5 PRP Transport Layers 25
3.6 PRP Action Reference. 29

Chapter 4. PMD C Language Library Procedures 77
4.1 Naming Conventions 77
4.2 Data Types 77
4.3 Return Values 78
4.4 C-Motion Engine Macros. 78
4.5 PMD Library Procedures 79

Chapter 5. C-Motion Engine 123
5.1 C-Motion Libraries. 123
5.2 Procedures Specific to the C-Motion Engine 123
5.3 C-Motion Engine Programming 124
5.4 The Console. 127
5.5 User Packets 127
5.6 Exceptions 127

Chapter 6. C-Motion 129
6.1 C-Motion Versions. 129
6.2 Axis Handles 129

Chapter 7. VB-Motion 133
7.1 Visual Basic Classes 133
7.2 Using A Magellan Attached Device. 135
7.3 Error Handling. 136

Index 137

This page intentionally left blank.

1. Introduction

1.1 PMD Resource Access Protocol Overview

The PMD Resource access Protocol (PRP) is used to communicate with PMD products incorporating a C-Motion Engine (CME), as well as products with Ethernet interfaces. These products include the Prodigy/CME family of motion control cards, the ION/CME digital drive, and the Ethernet/Serial ION digital drive.

PRP provides several key features in the devices where it is used:

- A wide range of hardware resources may be addressed, for example, multiple on-card communication peripherals and C-Motion Engines.
- The PRP addressing scheme allows general networking, for example, it allows PRP devices to act as network bridges.
- Allows extension to new classes of devices and device resources, and allows easy interfacing with user-written C-Motion Engine programs.

1.2 Scope

This manual documents the software interfaces and binary protocols used for control of all PRP-based devices, including Prodigy/CME and ION/CME, as well as all PMD Ethernet devices. Collectively these will be called “PRP devices”.

The procedures and data types documented here are used both for programs running on host computers, and user programs running in the C-Motion Engine.

Software libraries in source form are provided to implement PRP communication on Windows® host computers. Library code is also provided for control of Magellan Motion Processors, either as part of a PRP device or separately, for example non-CME ION modules or non-CME Prodigy cards. This library code is called C-Motion, and its use is documented in the *Magellan Motion Processor Programmer's Command Reference.* In many cases there is a close correspondence between the C language library procedures and PRP messages that may make the PRP documentation useful for understanding how to control PRP devices, even for PRP device users who are not implementing a PRP interface library.

This page intentionally left blank.

2. PMD Resource Access Protocol (PRP) Tutorial

In This Chapter

- Resource Addressing
- Accessing the Communications Ports
- Accessing On-Card Resources
- Accessing Magellan-Attached Devices
- PRP Communication Formats

2.1 Resource Addressing

Host access to any of the Prodigy/CME cards or Ethernet-capable ION digital drives is provided by a protocol called the PMD Resource Access Protocol (PRP). This easy-to-use yet powerful system utilizes actions, resources, and addresses to access any of the functions of the card or digital drive. PRP is used for host communications using serial, CANbus, or Ethernet.

Various PRP device functions are organized into *resources*; resource process *actions* sent to them. Actions can send information, request information, or command specific events to occur. *Addresses* allow access to a specific resource on the card, or connected to the card, via the PC/104 bus, serial, CANbus, or Ethernet connections.

A basic communication to a PRP device consists of a 16-bit PRP header, and an optional message body. The message body contains data associated with the specified PRP action; some actions do not require a message body. After a PRP communication is sent to a device, a return communication is sent by the PRP device which consists of a PRP header and an optional return message body. The return message body may contain information associated with the requested PRP action, or it may contain error information if there was a problem processing the requested action.

There are five different resource types supported by PRP devices. The ***Device*** resource indicates functionality that is addressed to the entire card or digital drive, the ***MotionProcessor*** resource indicates a Magellan Motion Processor, the ***CMotionEngine*** resource indicates the C-Motion Engine, the ***Memory*** resource indicates the dual-ported RAM or the non-volatile RAM (Random Access Memory), and the ***Peripheral*** resource indicates a communications connection.

There are ten different PRP actions including **Command**, which is used to send commands to resources such as the Magellan Motion Processor, **Send** and **Receive**, which are used to communicate using the serial, CANbus, and Ethernet ports, **Read** and **Write**, which are used to access memory-type devices such as the on-card dual-ported RAM, and the non-volatile RAM, and **Set** and **Get**, which are used to load or read parameters.

The remainder of this chapter, and Chapter 3, PRP Reference describe all of these constructs in more detail. More specific information on using the functions of a particular device is available in the *ION Digital Drive User's Manual*, the *Prodigy/CME PC/104 User's Guide*, the *Prodigy/CME PCI User's Guide* and the *Prodigy/CME Stand-Alone Card User's Guide.*

2.1.1 PMD Resource Access Protocol (PRP)

The core of the PMD Resource Access Protocol is a header that accompanies all PRP communications. Figure 2-1 shows the format of the resource access protocol header. The PRP header is a single 16-bit word divided into five fields. Normally, the PRP header is immediately followed by a message body, but there are certain communications that do not require a message body. PRP headers are used in both the outgoing, and the response packet. The returned PRP header is 1 byte in length, consisting of the version field, the status code field, and 4 reserved bits.

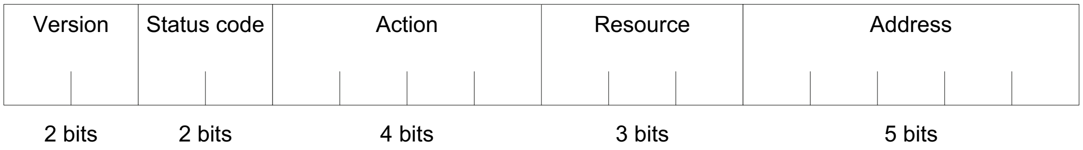
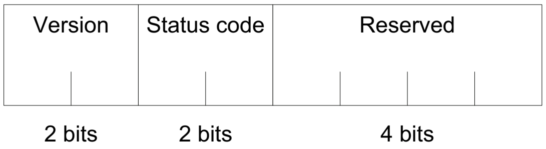


Figure 2-1: Outgoing and Returning PRP header formats

Return PRP Header



The majority of communications to/from the host controller and the PRP device use the PRP header. Exceptions are 'low-level' communications sent or received directly from the Serial, CANbus, or Ethernet ports. See Section 2.2, Accessing the Communications Ports, for more information.

PRP header field descriptions:

Version - This two bit field encodes the version of PRP being used. The value of this field for all PRP devices should always be 1 (binary 01) unless documentation included with your PRP device indicates otherwise.

Status code - For PRP commands being sent out, this 2-bit field should contain the value 2. When received, a return value of 0 indicates that this message is a normal response to an outgoing PRP command, a return value of 1 indicates that an error occurred during PRP command processing, and a value of 3 indicates that this is an asynchronous event message originated by the PRP device or by a device attached to the card. Each of these different response status codes may have information loaded in the PRP message body. See the *PMD Resource Access Protocol Programmer's Reference* for more information.

Action - This 4-bit field contains an action identifier that is used to process PRP messages. See Section 2.1.3, PRP Actions, for a summary of the PRP actions supported by the PRP device. This field is not used in the return PRP header.

Resource - This 3-bit field encodes the specific resource being addressed. See the table in Section 2.1.2, PRP Resources, for the complete resource map of the PRP device. This field is not used in the return PRP header.

Address - This 5-bit field encodes the address of the particular resource being communicated to. Fixed addresses are used for resources that are local to the PRP device. See the table in Section 2.1.2, PRP Resources, for a resource map of these addresses. Automatically assigned addresses are used to access attached devices, and are also used to create peripheral connections, which are communication 'conversations' between the PRP device and another device. This field is not used in the return PRP header.

The following sections provide general information on the PRP system. For a detailed description of the PRP header, resources, and supported actions, see Chapter 3, PRP Reference.

2.1.2 PRP Resources

PRP devices may support five different resource types, each of which is identified by a specific resource ID. In addition each available resource has an address. The following table summarizes these resource IDs and addresses for the on-card PRP device resources:

Resource Name	Resource ID	Address	Comments
Device	0	0	The Device resource indicates the PRP device itself.
CMotionEngine	1	0	The CMotionEngine resource indicates a C-Motion Engine.
MotionProcessor	2	0	The MotionProcessor resource indicates a Magellan Motion Processor.
Memory	3	0	The Memory resource indicates a dual-ported random access memory (RAM) or non-volatile RAM (NVRAM).
Peripheral	4	0	The Peripheral resource indicates a communications connection. A peripheral address of 0 indicates a 'null' peripheral. See Section 2.2.1, Peripheral Connections, for more information.

2.1.3 PRP Actions

The PRP device supports ten different actions, each of which is identified by a specific Action ID. In addition, many actions have sub-actions associated with them that are loaded into the PRP message body. For more information on actions, sub-actions, and the exact format by which the message body should be loaded, see Chapter 3, PRP Reference. The following table summarizes the action IDs for the PRP devices:

Name	Action ID	Used by Resource	Comments
NOP	0	All	The No Operation (NOP) command is used to verify connection to a given resource.
Reset	1	Device, MotionProcessor	Resets the specified resource.
Command	2	MotionProcessor, CMotionEngine	Sends a command to the specified resource.
Open	3	Peripheral, Device	Creates a new addressable resource. Resource addresses created using the Open action are not fixed, they are assigned automatically at the time a connection is requested.
Close	4	Peripheral, Device, MotionProcessor	Closes a resource created by Open, freeing the automatically assigned address. Used when a resource is no longer needed.
Send	5	Peripheral, C-Motion Engine	Sends a message to the specified resource.
Receive	6	Peripheral, C-Motion Engine	Receives a message from the specified resource
Write	7	Memory	Writes a data word to a memory resource
Read	8	Memory	Reads a data word from a memory resource
Set	9	Device, CMotionEngine	Sets parameters for a specified resource
Get	10	Device, CMotionEngine	Get parameters for a specified resource

2.2 Accessing the Communications Ports

Every PRP device uses a high-speed bus for internal communication. This bus interconnects resources within the device, such as the Magellan Motion Processor, the C-Motion Engine, and the Dual-ported RAM. External communications use this same bus to access communication ports such as serial, CANbus, or Ethernet. This means that commands to device resources may originate from outside the device via the serial, CANbus, or Ethernet ports, or internally via the C-Motion Engine.

A powerful feature of the PRP device's communication bus is that it can process multiple communication requests simultaneously. For example the C-Motion Engine can be used to send motion commands to the on-card Magellan Motion Processor, while an external host controller can communicate to the same Magellan via the Ethernet port to monitor progress of the moves, and display results on a remote capture/analysis program, such as PMD's Pro-Motion software.

While it is allowed to have more than one communications channel send motion commands to the PRP device's Magellan Motion Processor and another channel send monitoring-related requests, it is not advisable to have multiple communication channels send motion commands to the Magellan at the same time. This may result in unexpected or unsafe motion.

2.2.1 Peripheral Connections

PRP supports four different programmable network connection types, PC/104 bus, Serial, CANbus, and Ethernet. Every PRP device provides one or more of these interfaces. These communication resources are represented in PRP by a construct called a peripheral connection. A peripheral is a resource (resource ID: 4), and is provided, or utilized, by various PRP actions to send and receive messages to network connections.

Obtaining access to either the PC/104 bus, serial, CANbus, or Ethernet port is accomplished via the PRP Open action. This action opens a peripheral by specifying a sub-action of OpenISA, OpenSerial, OpenCAN, OpenTCP or OpenUDP, as well as the detailed connection parameters that will be used during communications with that specific peripheral connection. Each newly opened peripheral connection receives an automatically assigned address. The application code that requests the new peripheral connection must record that provided address for future use, and it is this address that is used within the PRP message to reference the newly created peripheral connection.

Automatically assigned addresses generally increment by one each time they are assigned, however this should not be assumed. The exact return address value should be stored and only that value should be used to reference that specific peripheral. Each ***Peripheral*** Open action is specific to the type of connection being requested. For example there is an 'open peripheral' command for CANbus communications, one for Serial communications, and so on.

To send a message to the new peripheral connection, a timeout parameter and the desired message is loaded into the PRP message body and the Send action is specified. To retrieve messages from that peripheral connection the Receive action is used. The Receive action requires that an amount of time (the communication timeout) be loaded into the message body. If a message is not received in the specified amount of time an error is returned.

Example

Figure 2-2 shows a network configuration. The host controller needs to initiate, send and receive a message to/from a specific device connected on the CANbus port.

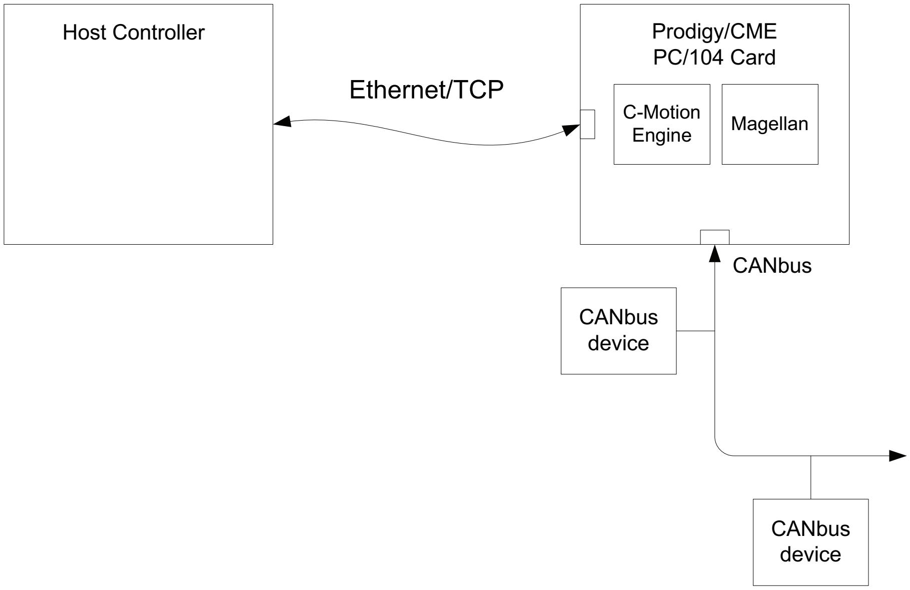


Figure 2-2: Example Network Configuration

This sequence will be accomplished in three steps. PRP messages will be assembled that comprise the **Open** action, and then in turn the **Send** and **Receive** actions. As for all PRP messages, the resource ID, the address, and the Action ID must be specified. In this example, to start the sequence the Resource is a 4 specifying a ***Peripheral***, the Action ID for **Open** is 2, and the PRP address will be loaded with a 0. The message body is loaded with a code for the sub-action **OpenCAN** as well as the detailed CANbus communications parameters to establish the connection, such as baud rate, send address, and receive address. For the exact message body of this and all PRP actions, refer to Chapter 3, PRP Reference.

> To simplify the nomenclature for PRP messages, a shorthand will be used which contains the mnemonic 'PRP,' the Resource ID, the string 'Addr,' the resource address, and the PRP action. Any additional information pertaining to the error code or message body will be contained in the comment for the message.

```
NewPeriphID = PRP Device, Addr 0, Open     // Send a request to open a connection to the device being
                                           // addressed via the CANbus port.
                                           // The Message body contains the detailed CANbus parameters
                                           // for that connection.
                                           // The Address of the new Peripheral connection is contained in
                                           // the body of the return message.
PRP Peripheral, Addr NewPeriphID, Send     // Send a message, contained in the message body, to the CAN
                                           // bus connection created using the Open command. Note that
                                           // the Peripheral address provided in the previous command is
                                           // used to identify which Peripheral connection the message
                                           // should be sent to.
PRP Peripheral, Addr NewPeriphID, Receive  // Receive a message. The string will be contained in the message
                                           // body of the return message from the CANbus Peripheral
                                           // connection.
```

Sending a message to the ***Peripheral*** resource at address 0 has special meaning as the 'null' peripheral. This peripheral address indicates 'no peripheral,' and may be useful in certain situations for disabling console messages, or disabling the Magellan's event output mechanism.

2.2.2 Managing Automatically Assigned Addresses

Although most network configurations are created once and left in place while the machine is operating, there may be circumstances where peripheral connections are made, and are then no longer needed. If this is the case, these connections should be 'closed,' thereby freeing that automatically assigned ***Peripheral*** address. This is accomplished via the PRP action Close.

Figure 2-3 shows a PRP device being used as a production Ethernet device exerciser.

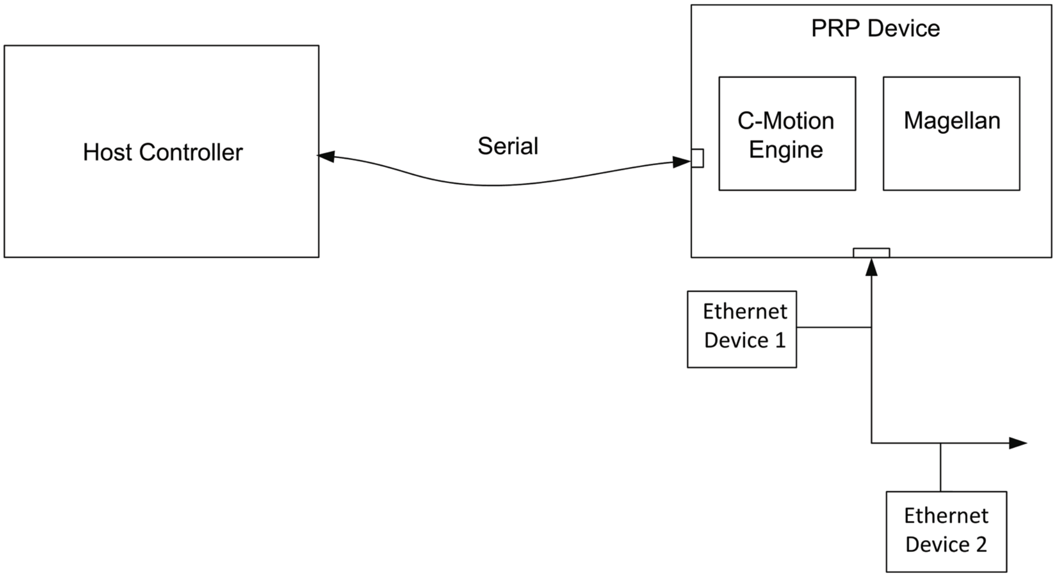


Figure 2-3: Example PRP Device Architecture with Ethernet Device Testers

In this example the ION/CME module sends messages to two different Ethernet/TCP addresses and then closes those peripheral connections to allow the process to be repeated indefinitely.

```
PeriphID1 = PRP Device, Addr 0, Open        // Open a peripheral connection to Ethernet device 1
PRP Peripheral, Addr PeriphID1, Send        // Send a test string to Ethernet device 1
PeriphID2 = PRP Device, Addr 0, Open        // Open a peripheral connection to Ethernet device 2
PRP Peripheral, Addr PeriphID2, Send        // Send a test string to Ethernet device 2
PRP Peripheral, PeriphID2, Close            // Close peripheral connection for Ethernet device 2
PRP Peripheral, PeriphID1, Close            // Close peripheral connection for Ethernet device 1
```

For complete information on the format and function of these, and other PRP commands, refer to Chapter 3, PRP Reference,.

2.3 Accessing On-Card Resources

The most common use of any PRP device is to process commands to resources located on the device itself. Typically this means communicating with the on-card Magellan Motion Processor, but it can also mean communicating with other device resources such as the dual ported RAM. Figure 2-4 shows this configuration.

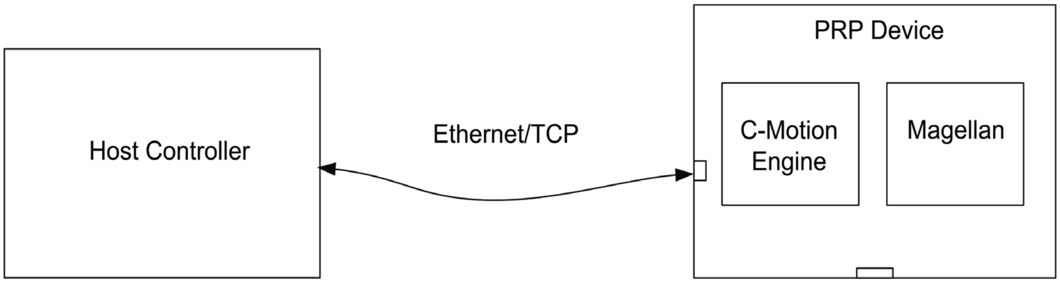


Figure 2-4: Host Controller & On-card Resources

Accessing on-card resources is straightforward using the PRP system. The on-card resource and the address are looked up using the table in Section 2.1.2, PRP Resources, and then a PRP message corresponding to the desired action is sent using those on-card resource and address values. The following example illustrates:

Example 1: A Host Controller wants to set the position of Axis 3 of the PRP device's on-card Magellan Motion Processor to a value of 0x123456.

From the table in Section 2.1.2, PRP Resources, to communicate with the on-card Magellan Motion Processor, a PRP message is sent to Resource ID 2 (corresponding to the ***MotionProcessor*** resource), to address 0 (corresponding to the PRP device's on-card Magellan), with an action ID of 2 (corresponding to the Command action). The message body is loaded with the Magellan packet corresponding to "Set Position, #3 0x123456," which is the 3-word sequence 0x210, 0x0012, 0x3456.

```
PRP MotionProcessor, Addr 0, Command   // Send a command to the on-card Magellan Motion Processor. Message
                                       // body contains Magellan Command "SetPosition #3, 0x123456"
```

Upon processing of this command, the host would receive a PRP message back. A zero in the status field would indicate that no error occurred. If this is the case the message body will be empty. If an error did occur, then the PRP status field would contain a 1, and the message body would contain the specific error code that occurred.

Example 2: The Host Controller reads the 32-bit word value of address 0x100 from the PRP device's dual ported RAM

Here is the command that would be sent:

```
PRP Memory, Addr 0, Read               // Send a 'Read' action (ID: 7) to the PRP Memory Resource (ID: 3),
                                       // address 0 (the address of the PRP device on-card dual
                                       // ported RAM). The PRP message body contains a sub-action of 0,
                                       // specifying a 32 bit word read, followed by 0x100 the address of the
                                       // 32 bit word to read from the dual ported RAM.
```

Upon successfully processing this command, the host would receive the 32-bit contents of memory location 0x100 in the message body.

Note that the PRP message to send a Magellan command packet did not use a sub-action code in the message body, while the **Read** command sent to the dual ported RAM did. Whether or not a sub-action is required, and what the codes are for various sub-actions, is action-specific, and sometimes resource-specific. Chapter 3, PRP Reference provides exact message body information for each PRP action and (if applicable) sub-action.

Note also that in this discussion of sending PRP messages to the PRP device, the specific communications channel to the card (serial, CANbus, Ethernet) was not discussed. That is because the PRP header and message body are the same regardless of how transmission occurs. Depending on the network type used, those varying communication links will send, receive, and process errors for packet communications in different ways, but from the perspective of the PRP system, those message transmission details are handled automatically. This is a very powerful characteristic of the PRP system, and we will see additional examples of similar 'access virtualization' as we discuss more advanced network topologies.

To actually send a PRP command over a serial, CANbus, or Ethernet network, specific protocols must be observed. See Section 2.5, PRP Communication Formats, for this network-specific (serial, CANbus, or Ethernet) information.

2.4 Accessing Magellan-Attached Devices

Section 2.2.1, Peripheral Connections, provided information on how general purpose messages can be sent to, or received from, any of the PRP device's network ports. This level of low-level access can be useful to communicate with a wide variety of custom-created or off-the-shelf products that are capable of communicating on that bus.

If the attached device is a PMD device however, such as a non-/CME ION or non-/CME Prodigy PCI card (PR825xx20 family and PR925xx20 family) then it is possible to integrate these devices into the PRP access network so that they can be communicated to without using low-level peripheral send and receive commands.

IONs and non-/CME Prodigy PCI cards are referred to as 'Magellan-Attached' devices because the network directly connects to the Magellan Motion Processor, and utilizes the Magellan's own communication protocols for receiving commands and returning data along the communication link.

Figure 2-5 provides an example of a setup where a Prodigy/CME PC/104 card is connected via Ethernet to a host controller, and where a second non-CME Prodigy PC/104 card is attached to the PC/104 bus.

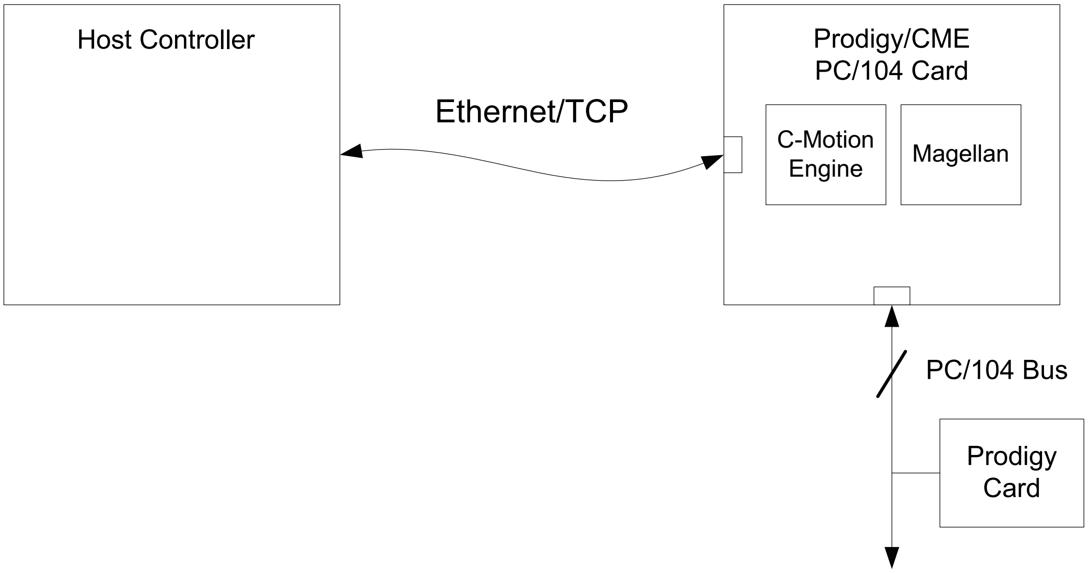


Figure 2-5: Host Controller & Magellan-attached Devices

In this 'bridge' configuration, the host controller (or the PRP device's C-Motion Engine) can access the PC/104-connected Prodigy card very similarly to the way in which the on-card Magellan is addressed. To accomplish this however, a different method is used to create access to the Magellan compared to the method described for on-card Magellan access in (see Section 2.3, Accessing On-Card Resources, for details).

Rather than sending PRP messages to the on-card ***MotionProcessor*** resource, first a raw peripheral connection is opened, and then the Open action with a sub-action of MotionProcessor is used to open a new ***MotionProcessor*** resource. The following PRP code sequence illustrates:

```
NewPeriphID = PRP Device, Addr 0, Open      // Open an ISA peripheral connection to connect to the slave
                                            // Prodigy-PC/104 card
NewMtnProcID = PRP Peripheral, Addr NewPeriphID, Open
                                            // Use the opened peripheral connection to create a new
                                            // MotionProcessor resource using the Open action with
                                            // sub-action specifying MotionProcessor. A new MotionProcessor
                                            // resource address is loaded into the message body
PRP MotionProcessor, Addr NewMtnProcID, Command
                                            // Send Magellan command packet to the slave Prodigy-PC/104's
                                            // Magellan Motion
                                            // Processor using the standard method of communicating with
                                            // MotionProcessor resources
```

Note that as the command sequence above shows, after the new ***MotionProcessor*** resource addresses are created using the Open action, subsequent Magellan commands to the slave Prodigy-PC/104 are identical in format to commands to the on-card Magellan. This illustrates a very powerful feature of the PRP system which is that it allows resources to be addressed transparently by the host controller (or C-Motion Engine module), making it easy to create and access networks of PMD products.

Note also that the Open action can be used with different resource types. In Section 2.2.1, Peripheral Connections, it was used with a resource type of ***Device*** to open a peripheral connection. In the above example it was used with a resource type of ***Peripheral*** to open a connection to a Magellan Motion Processor.

2.5 PRP Communication Formats

The following sections discuss how PRP messages should be formatted on the Serial, CANbus, and Ethernet networks so that they can be properly interpreted by the PRP device. Note that PRP messages from a host are not sent over the PC/104 bus, because the only PC/104 PRP device, the Prodigy/CME PC/104 card, acts as the bus master. Communication via PRP from a host to the PRP device occurs via serial, CANbus, or Ethernet.

2.5.1 PRP Messages Over Serial

PRP devices can receive PRP command messages from a host controller on one or more serial ports. This section describes the format of these packets, which transfer PRP messages over a serial protocol.

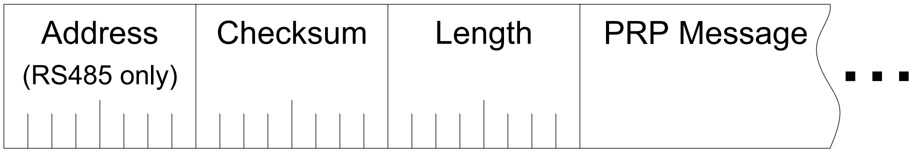
Figure 2-6 shows the Serial packet protocol that must be used for all PRP messages. This serial header is 32 bits, and is broken into three fields as follows:

Address: the first byte is an address field, used only with RS485 communications. For RS232 communications, this byte is not included in the packet.

Checksum: The second byte is a simple 8-bit additive checksum of all bytes in the packet payload (excluding the serial header, i.e., the address, checksum, and length fields). From Figure 2-6 this means only the PRP header and message body contribute to this checksum value

Length: The third byte is an 8 bit length field. The length indicates the number of bytes in the packet payload. The packet payload is defined as everything in the serial packet after the length field. That is, the 2 byte PRP header plus however many bytes are contained in the PRP message body. For example if the PRP message body is 6 bytes in size, the value filled into the length field should be 8 (2 bytes for PRP header + 6 bytes for the PRP message body).

Figure 2-6: PRP Message over Serial Format



Return PRP messages should be formatted in the same way.

2.5.1.1 Serial Packet Processing

An error-free Serial/PRP communication sequence from the host controller to the PRP device consists of a full outgoing packet transmission with the correct checksum and specified number of bytes received by the PRP device, and a full packet response with correct checksum and length received at the host controller. The return message must be received within a fixed amount of time determined by the host controller. Correctly setting this 'timeout window' may depend on factors such as baud rate, but 100 msec is a typical safe value.

If the host controller receives a response packet with an incorrect checksum, or does not receive the complete response (communications timeout), then the original message should be resent.

If the PRP device receives a packet with an incorrect checksum, then a packet with an error code indicating this is returned to the host controller.

If the PRP device does not receive the specified number of bytes within 100 msec (the PRP device timeout value) of beginning of packet reception, the incoming message is ignored, and no message is sent to the host controller.

2.5.2 PRP Messages Over CANbus

If the PRP device is set up to process PRP command messages from a host controller over CANbus, a specific format for the packets must be followed. This section describes the format of how PRP messages are carried over CANbus.

Since native CANbus communications can not be larger than 8 bytes, hosting the PRP system, which can support shorter as well as longer messages, requires additional layers to manage data segmentation and desegmentation. The protocol that is used by the PRP devices to accomplish this is very similar to the Service Data Object (SDO) protocol of the CANopen standard.

The details of this protocol are extensive enough that they are not described here, but are available in the Chapter 3, PRP Reference.

2.5.2.1 CANbus Packet Processing

Unlike the serial protocols, the SDO-based CANbus protocol has a robust error checking and retransmission mechanism built in that corrects for garbled or otherwise unusable transmissions.

Nevertheless, if a host controller does not receive the complete response packet within a specific time window (communications timeout), then the original message should be resent.

2.5.3 PRP Messages over Ethernet

The existence of ports, and the broad range of packet lengths that are supported with the Ethernet TCP/IP protocol, makes sending PRP messages very simple. For both sent and received messages the PRP message is simply loaded as the ‘payload’ of the Ethernet message. The only convention that must be observed is that the host controller’s destination TCP port must be equal to the PRP device’s TCP default value set using the **SetDefault** command. Note that the UDP protocol may not be used for PRP communications to/from the PRP device.

2.5.3.1 Ethernet Packet Processing

Unlike the serial protocols, Ethernet TCP packets have a robust error checking and retransmission mechanism built in that corrects for garbled or otherwise unusable transmissions.

This page intentionally left blank.

3. PRP Reference

In This Chapter

The PMD Resource access Protocol (PRP) is used for sending commands to, and receiving status information from, a PRP device, such as an ION/CME module or Prodigy/CME card. The protocol may be transported using a serial line (RS232/485), Ethernet TCP/IP, CANbus, or PCI bus. The central idea of the PRP is that of commanding an *action* to a specific *resource* identified by an *address*, on a PRP device. The table in Section 3.4.1, Outgoing PRP Packet illustrates how these data are encoded in an outgoing packet.

In addition to commands made directly to a PRP device, PRP may be used to send commands to Magellan Motion Processors, for example non-CME ION modules and non-CME Prodigy cards, that are accessible through a Prodigy/CME card. These will be called "*Magellan attached*" devices. Finally, PRP may be used to route commands to a PRP device via another PRP device.

3.1 PRP Resources

A PRP *resource* is a category of object to which commands may be sent. Objects are grouped together whenever they may be used in the same fashion, for example all connections to remote devices over any sort of communication channel are grouped together as *peripheral* resources. Resources may be *local*, that is provided by a PRP device that is directly accessible from a host program, or *remote*, that is accessible only through another PRP device. Local resources have fixed addresses beginning with zero, while remote resources have addresses that are assigned by a PRP request to a Prodigy/CME device.

The following table summarizes the various resource types and their numeric codes.

Name	Code	Description
Device	0	A Prodigy/CME card or ION/CME module
CMotionEngine	1	A C-Motion Engine
MotionProcessor	2	A Magellan Motion Processor
Memory	3	A random access memory
Peripheral	4	A connection to a remote device over a communications channel.
Reserved	5-7	

Device means a Prodigy/CME board, ION/CME module or a future device that understands the same protocol.

CMotionEngine means the C-Motion engine of a **device**, a functional unit capable of running user-specified C-Motion programs.

MotionProcessor means a Magellan Motion Processor, which may be part of a Prodigy/CME card, a non-CME Prodigy card, an ION/CME module, a non-CME ION module or a user-designed device including a Magellan Motion Processor.

Memory means a memory unit capable of being randomly addressed. The dual-ported RAM available on Prodigy/CME boards is a ***memory***.

Peripheral means an open connection to some off-card object using a communications device. This resource type applies to connections made using TCP, UDP, CANbus, serial, or PC-104 ISA bus, as well as any similar transport mechanisms that may be supported in the future. The address zero is reserved for the *null peripheral.* Data sent to the null peripheral will be discarded without error, reading from the null peripheral will never produce any data, and will eventually time out.

3.2 PRP Addresses

Every resource accessible using PRP is identified by a numeric address. Addresses for resources local to a PRP device are fixed numbers beginning with zero, addresses for resources on remote PRP devices, that is devices not directly connected to the host, are obtained by PRP actions and are dynamically assigned.

While these remote addresses may in practice be predictable, it is important not to assume their values, which may change depending on the state of the device assigning them. Remote addresses are specific to the PRP device that assigned them, they represent the beginning of a path to the remote resource, not a global network address for it.

Although PRP supports general remote addressing, it is useful even when restricted to local addresses, and it is likely that many applications will use only local addressing.

3.3 PRP Actions and Sub-Actions

A PRP *action* is a particular operation, specified by a 4 bit integer. "Pocked" or "command" would have been reasonable names instead of "action." All the different variants of a given action are logically similar.

The behavior of an action depends on the resource type to which it is addressed, the same action may take a different set of arguments, return different data, and have different effects depending on its resource type. Many, but not all, actions are only fully specified by adding a *sub-action,* an 8-bit code qualifying the action to take. Finally, a few commands also accept a *sub-sub-action,* another 8-bit qualifier of the action to take.

An action is fully specified by the action, the resource, and the sub-action. In the detailed PRP action reference section the action appears on the left side of the heading, the resource in the middle, and the sub-action on the right.

The following table summarizes the action codes. See Section 3.6, PRP Action Reference for more information.

Name	Value	Meaning	Resources
NOP	0	No operation	All
Reset	1	Perform a reset	Device, MotionProcessor
Command	2	Motion Processor and miscellaneous actions	MotionProcessor
Open	3	Open an addressable resource	Peripheral, Device
Close	4	Close a remote resource	any remote resource
Send	5	Send data to a stream-like resource	Peripheral, CMotionEngine
Receive	6	Receive data from a stream-like resource	Peripheral, CMotionEngine
Write	7	Write data to an indexed resource	Memory, Peripheral (ISA)
Read	8	Read data from an indexed resource	Memory, Peripheral (ISA)
Set	9	Change a setting or operating state	Device, CMotionEngine
Get	10	Get a setting or operating state	Device, CMotionEngine
Reserved	11-15		

3.4 PRP Packet Structure

Every PRP command is sent as a data packet, and how the packet is structured depends on the transport mechanism used. In every case however, the receiver of the packet can determine its length. For serial and CANbus ports a PMD protocol is used to encode the length. For other transport mechanisms standard means are used to determine length. Every command packet begins with a two byte (16 bit) *header,* followed by a *message body,* the length of the message body depends on the particular action, some actions do not require a body, some a fixed length body, and some a variable length body.

There are four PRP packet types, each of which has a unique value of the *status* field in the PRP header:

- Status 2: Outgoing (command) packets are used to request information or command a PRP device to do something.
- Status 0: A success response packet is returned after an outgoing packet that may be complied with. The response packet may include one or more data bytes depending on the particular action that was requested.
- Status 1: A failure response packet is returned after an outgoing packet that may not be complied with. A 16 bit error code is included to indicate the nature of the problem.
- Status 3: An asynchronous event packet is sent from a PRP device to a host in response to a Magellan motion processor event that has been selected using the Magellan **SetInterruptMask** command, documented in the “Host Interrupts” section of the *Magellan Motion Processor User’s Guide.* Event notifications from Prodigy/CME Magellan Motion Processors or from Magellan-attached devices may be propagated back to a host computer if all of the peripheral connections were opened with appropriate settings for handling events.

3.4.1 Outgoing PRP Packet

The table below shows the structure of an outgoing PRP packet:

Outgoing PRP Packet

	7	6	5	4	3	2	1	0
header byte 0	version (1)		status (2)		action			
header byte 1	resource			address				
body byte 0…								

The action, resource, and address fields are explained in the previous sections.

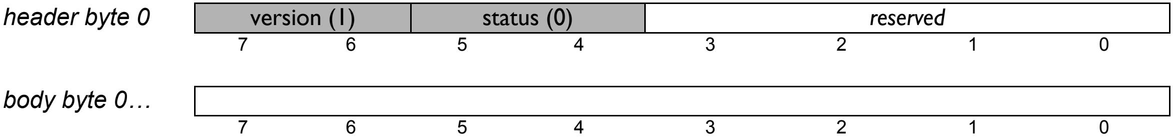
Shading is used for the version and status fields to indicate that their values are fixed. The version field is used for sanity checking and possible future protocol changes, for all current PRP devices the version field must contain 1. The **status** field is used in the response packet to indicate success or failure, and to distinguish outgoing from incoming packets. For all outgoing headers the status field must contain 2.

3.4.2 PRP Response Packets

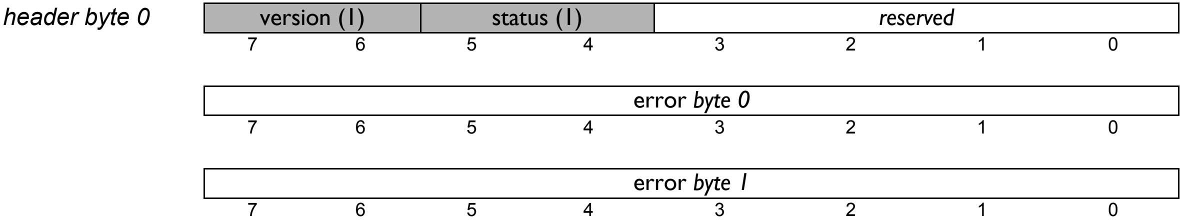
A PRP device must respond with a response packet, which has at least a one byte (8 bit) header, followed by a message body. The length of the message body depends on the particular action - in some cases no body is required, in some cases a fixed length body is required, and in some cases a variable length body is used. In the case of a variable length body, information on packet length external to PRP must be used to determine the length.

The table below shows the structure of PRP response packets for success and for failure:

PRP Success Response Packet



PRP Failure Response Packet



The version field, as for the outgoing packet, must contain 1.

The bits marked *reserved* may have any value, may be unpredictable, and should be ignored.

The status field is used to indicate success or failure, a value of zero indicates success, and a message body may follow as specified by the documentation for the particular action to which the PRP device is responding. A **status** value of 1 indicates that an error occurred processing the requested action, and a two byte (16 bit) message body follows specifying the particular error that occurred. The table below summarizes the values that the error code may take. When used in the C language interface these names should be prefixed by "**PMD_ERR_RP_**," for example, "**PMD_ERR_RP_InvalidAddress**."

Name	Value	Description
Reset	0x2001	The previous command reset the device; action was not processed.
InvalidVersion	0x2002	The version field was incorrect.
InvalidResource	0x2003	No such resource type.
InvalidAddress	0x2004	The address for the specified resource type is not valid.
InvalidAction	0x2005	No such action, or resource not appropriate to specified action.
InvalidSubAction	0x2006	Sub-Action field not valid, or resource not appropriate for sub-action.
InvalidCommand	0x2007	An enumerated option argument is not correct.
InvalidParameter	0x2008	An argument value is not legal, or not supplied.
InvalidPacket	0x2009	A PRP packet was corrupted
Reserved	0x200A-0x200D	
Checksum	0x200E	Bad packet checksum value
Reserved	0x200F-0xFFFF	
Magellan error codes	1 – 18	Magellan Motion Processor error codes, documented in the *Magellan Motion Processor User's Guide*.

3.4.3 Event Notification Packet

Asynchronous events may be signaled by a Magellan Motion Processor, either a Magellan attached device, such as an ION module, or a motion processor that is part of a PRP device. News of asynchronous events is propagated as promptly as possible to all devices with an open connection to the motion processor signaling them, so that user programs may check for them using the **PMDWaitForEvent** procedure without the network latency required for a Magellan command read of the event status register.

Asynchronous event notification is optional. In order for the event mechanism to work an appropriate event channel must be specified when opening each peripheral used for motion processor or PRP device communication. The event mechanism is supported by TCP/IP (see the **OpenTCP** action), PCI (see the **PMDPeriphOpenPCI 7** procedure), PC-

104 (see the **OpenISA** action), and CANBus (see the **OpenCAN** action). Serial peripherals do not support event notification.

The table below indicates the structure of an asynchronous event notification packet:

Event Notification PRP Packet

	7	6	5	4	3	2	1	0
header byte 0	version (1)		status (3)		eventType			
header byte 1	resource			address				
body byte 0	Axis							
body byte 1	EventStatus *byte 0*							
body byte 2	EventStatus *byte 1*							

An asynchronous event notification may be sent by a PRP device to a host at any time, so software to deal with PRP connections must be prepared to handle either an event notification packet or a response packet. The details of event packet transmission depend on the transport mechanism. Over TCP/IP connections event packets are sent to the same port and in the same way as response packets. Serial connections do not support event notification. PCI bus PRP connections use an interrupt and a separate memory buffer for event notification. For details on a specific transport mechanism see Section 3.5, PRP Transport Layers.

The **version** field, as for other packet types, must contain 1. The **status** field value of 3 identifies the packet type as event notification.

The **eventType** field identifies the sort of event signaled, currently only one type is supported, 2, meaning a Magellan Motion Processor event notification. All other values through 15 are reserved for future expansion, and it is expected that some will be used for PRP/CME-specific events.

The **resource** and **address** fields identify the specific resource that was responsible for signaling an event. In the case of a motion processor event, **resource** will be ***MotionProcessor*** (2). In the most common case of an event raised by a PRP device motion processor **address** will be zero, however for the case of a PRP device controlling a Magellan attached device, **address** will be the value assigned during the **Open Device OpenMotionProcessor** action.

The number and meaning of the body bytes will depend in general on the event type, but because only one event type is supported at this time its body layout is illustrated.

Axis indicates the motion processor axis signaling the event, zero meaning axis 1, one axis 2, and so forth.

EventStatus is the value of the 16-bit event status register at the time the event was signaled. The "Host Interrupts" section of Magellan Motion Processor User?s Guide documents the meaning of the bits of this register.

3.5 PRP Transport Layers

This section discusses the communication device-specific aspects of sending and receiving PRP packets.

PRP may be sent using a serial, PCI, CANbus, or TCP/IP communication channel. PRP is not currently supported over PC/104 because the only PC/104 PRP device, Prodigy/CME, may only be a PC/104 bus master.

3.5.1 PRP Serial Transport

A PMD protocol header is used to specify the length of PRP packets sent over a serial line, and to detect most cases of packet corruption. The Prodigy/CME card and ION/CME module are set up to receive serial PRP command messages from a host controller on Serial port 1.

There are two cases of the serial protocol:

1. Point-to-point serial communication using either RS232 or RS485: only one PRP device and one host may be connected to the serial line.
2. Multi-drop serial communication using RS485: multiple PRP devices may share the same serial bus, but each must be configured to use a separate multi-drop address.

The figures below illustrate the two cases:

Point-to-Point Serial Packet

7	6	5	4	3	2	1	0
checksum							
length							
PRP packet byte 0							
PRP packet byte 1 …							

Multi-Drop Serial Packet

7	6	5	4	3	2	1	0
MultiDropAddress							
checksum							
length							
PRP packet byte 0							
PRP packet byte 1 …							

The MultiDropAddress field is used to address a particular PMD serial device, and each device must be configured to use a different address.

The length field is the unsigned number of bytes in the PRP packet, put another way, for point-to-point packets there are length + 2 bytes in a serial packet, and for multi-drop packets there are length + 3.

The checksum field is a simple additive checksum modulo 256, over the bytes in the PRP packet, exclusive of the serial header. A checksum over all bytes in the serial packet excluding the MultiDropAddress and length bytes should be zero.

Both outgoing and response packets are formatted in the same way. Asynchronous event notification is not supported using serial transport.

An error-free Serial/PRP communication sequence from the host controller to the PRP device consists of a full outgoing packet transmission with the correct checksum and specified number of bytes sent to the card, and a full packet response with correct checksum and length received at the host controller. The return message must be received within a fixed amount of time determined by the host controller. Correctly setting this 'timeout window' may depend on factors such as baud rate, but 100 milliseconds is a typical safe value.

If the host controller receives a response packet with an incorrect checksum, or does not receive a complete packet (communications timeout), then the original message should be resent.

If a PRP device receives a packet with an incorrect checksum, then it will respond with an error response packet with an error code of PMD_ERR_RP_Checksum.

If the PRP device does not receive the specified number of bytes within 100 milliseconds of beginning of packet reception, the incoming message is ignored, and no message is sent to the host controller.

3.5.2 PRP TCP/IP Transport

PRP packets are realized as TCP/IP packets, except that three bytes of padding must be added before each outgoing (host to PRP device) packet. The padding bytes are ignored.

The length of each PRP packet is determined from the IP header.

In order to initiate a PRP connection, a host should establish a TCP connection to a PRP device using the port specified by the device default DefaultTCPPort. The factory default for this port is 40100, but it may be changed using **Set Device SetDefault**.

The Prodigy/CME and ION/CME module support asynchronous event notification over TCP/IP by sending event packets to DefaultTCPPort +1. The packet format is the same as the CAN event notification packet.

Word 1	Event axis
Word 2	Event status

PRP over TCP/IP supports asynchronous event notification by sending event packets over the same TCP connection as response packets, so the host must be prepared to read a packet at any time, and to distinguish event from response packets.

3.5.3 PRP PCI Bus Transport

The Prodigy/CME-PCI card uses the PCI 9030 interface chip from PLX technology. For information on the operation of this device or on developing driver software please refer to the PLX documentation available from www.plxtech.com.

The PCI PRP transport uses an on-card buffer mapped into PCI memory space for exchanging PRP packets. Two base address registers (BARs) found in the PCI configuration space for the card are used, BAR 2 is used for some control and status bits, and BAR 5 is used for reading and writing the PRP messages.

There are six base address registers in total, the actual base addresses are obtained from the host operating system or from the PLX driver:

- BAR 0 and BAR 1 are reserved by PLX.
- BAR 2 is used for command and status registers to control the flow of PRP messages and console data, including the registers illustrated in the table below. All registers are 16 bits wide.
- BAR 3 is used for resetting the card.
- BAR 4 is reserved.

- BAR 5 contains a 2048 byte on-card buffer for exchanging PRP packets and console data, illustrated in the table below:

BAR 2 Offset	Register	Description
0 – 0x000F	*Reserved*	
0x10	PCI_CMD_REG (read/write)	Command Register – contains the type of PCI transaction to be initiated. A write to this register initiates a PCI transaction. Bits 15:1 – Not used Bit 0 – set means PCI read(card to host); clear means PCI write (host to card)
0x12	PCI_STATUS_REG (read only)	Status Register – contains the status of the last PCI transaction. Bits 15:1 – Not used Bit 0 – set means PCI transaction complete. This nbit is celared by a write to PCI_CMD_REG.
0x14	PCI_IRQ_OUT_ENABLE (read/write)	Interrupt Enable Register – enables the corresponding individual PCI local bus interrupt requests.

BAR 5 Offset	PRP Message Buffer
0 - 3	PRP message length
4 - 0x3FF	PRP packet data
0x400 - 0x600	*Reserved*
0x600 - 0x7FF	Console message data

In order to send a PRP message over the PCI bus the following sequence should be followed:

1. Write the length to PCI BAR 5 offset 0, and the packet data beginning at PCI BAR 5 offset 4.
2. Set bit 0 in PCI_CMD_REG (BAR 2 offset 0x10) to inform the Prodigy/CME card that a PRP message is waiting.
3. Poll bit 0 in PCI_STATUS_REG (BAR 2 offset 0x12) until set by the card to indicate that a PRP response is ready.
4. Clear bit 0 in PCI_STATUS_REG.
5. Read the response length from PCI BAR 5 offset 0, and the response data beginning at PCI BAR 5 offset 4.

3.5.4 PRP CANbus Transport

PRP over CANbus uses the concept of a *node identifier*, a concept borrowed from CANOpen. The node identifier is a user-chosen integer between 1 and 127, inclusive, and is the least significant seven bits of any CAN identifier used for PRP communication. As long as their node identifiers are different, PRP devices should coexist (but not communicate) with CANOpen devices on the same CANbus.

PRP uses three CAN identifiers for communication:

- 0x600 + NodeIdentifier is used for sending messages from the host to a PRP device. This identifier is used by default for SDO transmit by CANOpen devices.
- 0x580 + NodeIdentifier is used for sending responses from a PRP device to a host. This identifier is used by default for SDO receive by CANOpen devices.

- 0x180 + **NodeIdentifier** is used for sending asynchronous event notification packets from a PRP device to a host.

CAN messages are limited to eight bytes of data, which means that many PRP packets require several CAN messages for transport. In order to support PRP over CANbus a segment/de-segment protocol is used. The first byte of each CAN message is used for segment information; all of the remaining bytes are used for the PRP header or body.

Each CAN message used for PRP is either an *initial* message, or a *continued* message. An initial message is the first message used for a PRP packet, and may be the only message in the packet. An initial message may be followed by zero or more continued messages, which complete the PRP packet.

The same protocol is used for sending PRP actions, responses, and event notification packets.

The header byte of each initial message has the form:

1	NContinued						
7	6	5	4	3	2	1	0

NContinued is the number of continued messages that will follow, and may be zero.

Each continued header byte has this form:

0	Sequence						
7	6	5	4	3	2	1	0

The first continued message has a **Sequence** value of one, the second two, and so forth. Each message has a **Sequence** value one greater than that of the previous message. The final message has a **Sequence** value of **NContinued**.

If a message is received with an unexpected **Sequence** value, or an Initial message is received when expecting a Continued message, then the receiver should immediately send a PRP error packet with the error code **CANMessageOutOfSequence** [NB. value and perhaps name TBD].

The exact length of a PRP packet may not be determined after reading just the initial message with a nonzero **NContinued** value, because the length of the last message is not known. The length is at least 7 * **NContinued** + 1 and at most 7 * (**NContinued** + 1).

No PRP packet checksum is required, because the integrity of each CAN message is protected by a CRC, naturally including the segment header bytes. Reception of the expected sequence numbers is very good evidence that a packet has been correctly received.

3.6 PRP Action Reference

This section describes each action and sub-action, with the binary encoding of all arguments. The following tables summarize the available actions and, where applicable, related C language procedures. The first table is arranged in alphabetical order; the second table is arranged in action code order.

Some aspects of action processing are common to all commands:

- Many PRP actions require a *sub-action* in addition to the action and resource, this is an 8-bit unsigned quantity that immediately follows the PRP outgoing header. Not all actions use a sub-action.
- The status field of a response packet is zero in case of successful command processing, and has the value 1 (Error) otherwise. In the error case the described returned data are not sent, instead a single 16 bit error code is sent in the response body. The reserved bits of a PRP response packet header may have any value, they are not guaranteed to be zero.
- The address field of a command header should hold a valid PRP address for the resource type sent. The address field of the response header will have the same value.

- A resource field that may have any of several values is indicated by the word resource, and the legal values specified in the resources section.
- All multi-byte argument values are encoded in little endian order: The least significant byte is sent first, and the most significant last. A 32 bit quantity is sent as bytes 0, 1, 2, and then 3, the most significant byte.
- Signed arguments are sent as twos-complement integers.

3.6.1 Action Table - Code Order

Action	Resource	Sub-action	C Procedure
NOP	*any*		
Reset	Device		PMDDeviceReset
	MotionProcessor	PMDDeviceReset	
Command	CMotionEngine	Flash	
		Task	PMDCMETaskStart
			PMDCMETaskStop
	MotionProcessor		*Any C-Motion Commands*
Open	Device	MotionProcessor	PMDAxisOpen
		CMotionEngine	PMDRPDeviceOpen
		Memory32	PMDMemoryOpen32
		Parallel	PMDPeriphOpenPAR
		ISA	PMDPeriphOpenISA
		COM	PMDPeriphOpenCOM
		CAN	PMDPeriphOpenCAN
		TCP	PMDPeriphOpenTCP
		UDP	PMDPeriphOpenUDP
	Peripheral	Device	PMDRPDeviceOpen
		MotionProcessor	PMDMPDeviceOpen
		MultiDrop	PMDPeriphOpenMultiDrop
Close	Peripheral		PMDPeriphClose
	Device		PMDDeviceClose
	MotionProcessor		PMDDeviceClose
	CMotionEngine		PMDDeviceClose
	Memory		PMDMemoryClose
Send	CMotionEngine		PMDPeriphSend
	Peripheral		PMDPeriphSend
Receive	CMotionEngine		PMDPeriphReceive
	Peripheral		PMDPeriphReceive
Write	Memory	Dword	PMDMemoryWrite
	Peripheral	Byte	PMDPeriphWrite
		Word	PMDPeriphWrite
Read	Memory	Dword	PMDMemoryRead
	Peripheral	Byte	PMDPeriphRead
		Word	PMDPeriphRead
Set	CMotionEngine	Console	
	Device	Default	PMDSetDefault
Get	CMotionEngine	Console	
		TaskState	PMDCMEGetTaskState
	Device	Default	PMDGetDefault
		ResetCause	PMDMBGetResetCause
		Version	PMDDeviceGetVersion

3.6.2 Action Table - Alphabetical Order

Action	Resource	Sub-action	C Procedure
Close	CMotionEngine		PMDDeviceClose
	Device		PMDDeviceClose
	Memory		PMDMemoryClose
	MotionProcessor		PMDDeviceClose
	Peripheral		PMDPeriphClose
Command	CMotionEngine	Flash	
		Task	PMDCMETaskStart
			PMDCMETaskStop
	MotionProcessor		*Any C-Motion Commands*
Get	CMotionEngine	Console	
		TaskState	PMDCMEGetTaskState
	Device	Default	PMDDeviceGetDefault
		ResetCause	PMDMBGetResetCause
		Version	PMDDeviceGetVersion
NOP	any		
Open	Device	CAN	PMDPeriphOpenCAN
		CMotionEngine	PMDRPDeviceOpen
		ISA	PMDPeriphOpenISA
		Memory32	PMDMemoryOpen32
		MotionProcessor	PMDAxisOpen
		COM	PMDPeriphOpenCOM
		Parallel	PMDPeriphOpenPAR
		TCP	PMDPeriphOpenTCP
		UDP	PMDPeriphOpenUDP
	Peripheral	Device	PMDRPDeviceOpen
		MotionProcessor	PMDMPDeviceOpen
		MultiDrop	PMDPeriphOpenMultiDrop
Read	Memory	Dword	PMDMemoryRead
	Peripheral	Byte	PMDPeriphRead
		Word	PMDPeriphRead
Receive	CMotionEngine		PMDPeriphReceive
	Peripheral		PMDPeriphReceive
Reset	Device		PMDDeviceReset
	MotionProcessor		PMDDeviceReset
Send	CMotionEngine		PMDPeriphSend
	Peripheral		PMDPeriphSend
Set	CMotionEngine	Console	
	Device	Default	PMDDeviceSetDefault
Write	Memory	Dword	PMDMemoryWrite
	Peripheral	Byte	PMDPeriphWrite
		Word	PMDPeriphWrite

Close *various*

Coding:

action	sub-action	resource
4	-	various

Arguments: none

Return Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		4			
write	resource			address				
read	2		status		*reserved*			

Description: The **Close** action may be used to free any resource that was originally returned by an **Open** action. After closing, such a resource no longer exists and will signal an error if an action is addressed to it.

Close will close an open TCP connection if applied to a TCP peripheral. For reasonably sized networks that are static it may never be necessary to use **Close**. It is an error to send a **Close** action to a resource that was not returned by **Open**.

C language syntax:

```
PMDresult PMDPeriphClose(PMDPeriph *periph);
PMDresult PMDDeviceClose(PMDDevice *device);
PMDresult PMDMemoryClose(PMDMemory *ram);
```

Command Flash CMotionEngine

Coding:

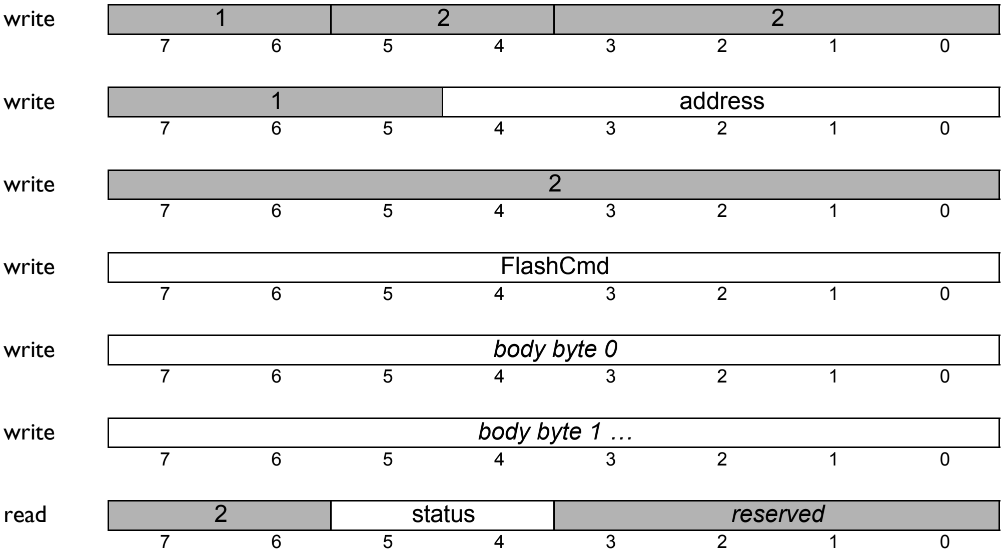
action	sub-action	resource
2	2	1

Arguments:

name	instance	encoding
FlashCmd	FlashStart	1
	FlashData	2
	FlashEnd	3

Returned Data: none

Packet Structure:


write
1
2
2
7 6 5 4 3 2 1 0
write
1
address
7 6 5 4 3 2 1 0
write
2
7 6 5 4 3 2 1 0
write
FlashCmd
7 6 5 4 3 2 1 0
write
body byte 0
7 6 5 4 3 2 1 0
write
body byte 1 …
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0

Description:

The **Command Flash CMotionEngine** action is used to install a user program in a C-Motion Engine. The flash process proceeds in three steps, each with a separate value of the *FlashCmd* argument. In addition to *FlashCmd*, this action may include many bytes of message body, depending on the step.

If any step of the flash procedure gives an error response then the procedure must be restarted from the beginning. No actions may be sent between flash procedure actions. The steps, in order of execution, are:

1. *FlashStart*: The body bytes are a four byte length of the flash image, least significant byte first. If this step is successful the user program flash is erased. The length may be specified as zero, in which case no new user program is installed, and no further steps need be taken.
2. *FlashData*: The body bytes are sequential parts of the entire flash image, in order.
3. *FlashEnd*: There are no body bytes. This action verifies the checksum of the program image received. If it finishes successfully then a new user program has been installed and may be run using the **Command Task CMotionEngine** action.

C language syntax:

The PMD C library does not support this operation. Pro-Motion may be used to flash user code images.

CommandTask CMotionEngine

Coding:

action	sub-action	resource
2	1	1

Arguments:

name	instance	encoding
option	1	start
	2	stop

Returned Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		2			

	7	6	5	4	3	2	1	0
write	1			address				

	7	6	5	4	3	2	1	0
write	1							

	7	6	5	4	3	2	1	0
write	option							

	7	6	5	4	3	2	1	0
read	2		status		*reserved*			

Description:

The **Command Task CMotionEngine** action is used to start or stop a C-Motion Engine user program. The two cases are distinguished by the argument **option.**

If ***option*** is **start**, then if a user program is currently running or if no user program is installed this action will return an error code.

If ***option*** is **stop**, then any running user program will be stopped. If no user program is currently running in the C-Motion Engine then this action will return an error code.

It is the user's responsibility to ensure safety when starting or stopping a user program that controls motors. It is not possible to predict the state of the PRP device or of its motion processor at the instant that the user program is stopped. PMD strongly recommends that a task be stopped only to correct unrecoverable errors and that the PRP device and any devices that it controls be put immediately into a known safe state using host commands. Because the card resources and the dynamic heap are not in a known state it is not safe to restart a task after stopping it without first resetting the entire device.

C language syntax:

```
PMDresult PMDCMETaskStart(PMDDeviceHandle *pDevice);
PMDresult PMDCMETaskStop(PMDDeviceHandle *pDevice);
```

Command MotionProcessor

Coding:

action	sub-action	resource
2	none	2

Arguments: Magellan command and arguments
rxCount, 2 bit count of words returned.

Return Data: Magellan return data

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		2			
write	2			address				
write	rxCount		*Magellan axis*					
write	*Magellan command code*							
write	*Magellan arguments …*							
read	2		status		*reserved*			
read	*Magellan data …*							

Description: The Command action directed to a ***MotionProcessor*** resource sends a Magellan protocol command to the motion processor indicated by the address field. A sub-action field is not used, instead a Magellan protocol command packet follows the header immediately.

Magellan commands are documented in the *Magellan Motion Processor Programmer's Reference Guide,* with the addition of the rxCount parameter. A Magellan protocol packet consists of at least one 16-bit command word, followed by zero to three argument words. The first byte of command word comprises two fields, bits 6 and 7 are the rxCount field, the number of words that are expected as returned values from the command. The remaining bits 0 – 5 are the Magellan axis addressed. The second byte of the command word is an opcode for the Magellan command. Each command takes a fixed number of arguments and returns a fixed number of return data. The arguments and data are encoded as big-endian quantities, in contrast to other PRP multi-byte arguments and data: 16-bit words are sent most significant byte first, followed by least significant byte, 32-bit words are sent in order of significance, starting with the most significant byte, and ending with the least significant.

If the status field of the return packet PRP header is zero then the return data of the Magellan command follow. If the Magellan motion processor reports an error then the status field of the return header will be 1 (error), and the Magellan error code will follow. Magellan error codes are documented in the *Magellan Motion Processor Programmer's Reference Guide,* and do not overlap with any PRP or PMD C library error codes. The error code will not be encoded as a big-endian value.

C language syntax: All C-Motion command procedures use this action. See the *Magellan Motion Processor Programmer's Guide* for documentation of C-Motion commands and C language syntax.

Get Console CMotionEngine

Coding:

action	sub-action	resource
10	4	1

Arguments: none

Returned Data:

name	type	meaning
Console	unsigned 8 bit	Peripheral address for console output

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		10			
write	1			address				
write	4							
read	2		status		*reserved*			
read	Console							

Description: The **Get Console CMotionEngine** action retrieves a peripheral address corresponding to a communications channel used for output of debugging and diagnostic messages by C-Motion user programs. The result of this action may not be meaningful if the console output was initially **Set** from a different device than the **Get** is issued from.

C language syntax: None, this action is not supported by the C library.

GetDefault Device

Coding:

action	sub-action	resource
10	2	0

Arguments:

name	type	meaning
DefaultCode	unsigned 32 bit	default identifier

Returned Data:

name	type	meaning
DefaultValue	varies	varies – see Set ValueDefault

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		10			
write	0			address				
write	2							
write	0							
write	DefaultCode *byte 0*							
write	DefaultCode *byte 1*							
write	DefaultCode *byte 2*							
write	DefaultCode *byte 3*							
read	2		status		*reserved*			
read	DefaultValue *byte 0*							
read	DefaultValue *byte 1 …*							

Description:

The **Get Default Device** action is used to retrieve the value of a device default. Device defaults are various non-volatile properties of the PRP device, for example the IP address, or whether to run a user program immediately after power up. The length of ***DefaultValue*** depends on the particular data type, and is encoded in the upper byte of ***DefaultCode***. A length value of zero means two bytes, one means four bytes. Please see the description of **Set Default Device** on page 70 for a table of supported default codes and their meaning.

C language syntax:

```
PMDresult PMDDeviceGetDefault(PMDDeviceHandle *hDevice,
                                          PMDDefault defaultcode,
                                          void *value,
                                           unsigned valueSize);
```

Note: At most value Size bytes will be written to the location pointed to by value.

Get ResetCause Device

Coding:

action	sub-action	resource
10	3	0

Arguments: none

Returned Data:

name	type	instance	encoding
ResetCause	unsigned 16 bit	0x0800	System Watchdog
		0x1000	hard reset
		0x2000	under voltage
		0x4000	external
		0x8000	watchdog

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		10			
write	0			address				
write	3							
read	2		status		*reserved*			
read	ResetCause *byte 0*							
read	ResetCause *byte 1*							

Description: The **Get ResetCause Device** action retrieves the cause of the last device reset.

C language syntax:

```
PMDuint16 PMDMBGetResetCause(PMDAxisHandle* axis_handle,
                             PMDuint16* resetcause)
```

see Please see the *Magellan Motion Processor Programmer's Reference* for procedure documentation.

Get TaskState CMotionEngine

Coding:

action	sub-action	resource
10	5	1

Arguments: none

Returned Data:

name	type	instance	encoding
TaskState	unsigned 8 bit	0	no program
		1	not started
		2	running

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		10			
write	1			address				
write	5							
read	2		status		*reserved*			
read	TaskState							

Description: The **Get TaskState CMotionEngine** action retrieves the current state of the user program in the C-Motion Engine addressed. Task states may be changed by using the **Command Task CMotionEngine** action.

C language syntax:

```
PMDresult PMDCMEGetTaskState(PMDDeviceHandle *pDevice,
PMDuint32 *state);
```

GetVersion Device

Coding:

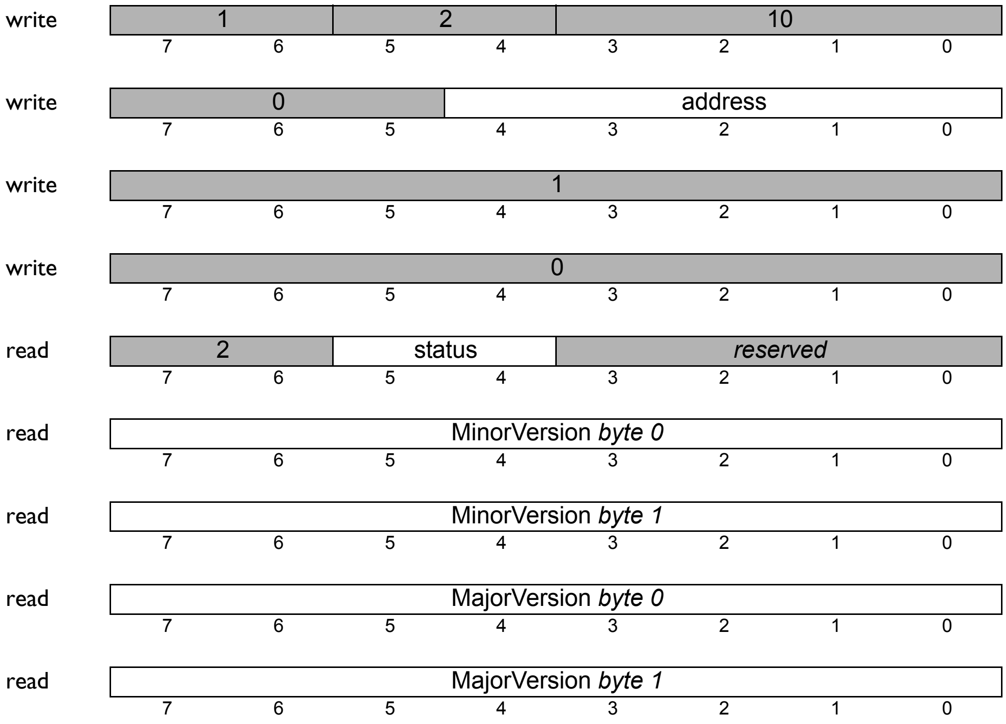
action	sub-action	resource
10	1	0

Arguments: none

Returned Data:

name	type	range
MajorVersion	unsigned 16 bit	0-0xffff
MinorVersion	unsigned 16 bit	0-0xffff

Packet Structure:


write
1
2
10
7 6 5 4 3 2 1 0
write
0
address
7 6 5 4 3 2 1 0
write
1
7 6 5 4 3 2 1 0
write
0
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
MinorVersion *byte 0*
7 6 5 4 3 2 1 0
read
MinorVersion *byte 1*
7 6 5 4 3 2 1 0
read
MajorVersion *byte 0*
7 6 5 4 3 2 1 0
read
MajorVersion *byte 1*
7 6 5 4 3 2 1 0

Description: The **Get Version Device** action retrieves version information for the PRP device addressed.

C language syntax:

```
PMDresult PMDDeviceGetVersion(PMDDeviceHandle *hDevice,
                              PMDuint16 *major,
                              PMDuint16 *minor);
```

NOP *any*

Coding:

action	sub-action	resource
0	none	any

Arguments: none

Return Data: none

Packet Structure:

write	1		2		0			
	7	6	5	4	3	2	1	0

write	resource			address				
	7	6	5	4	3	2	1	0

read	2		status		*reserved*			
	7	6	5	4	3	2	1	0

Description: The NOP action does not result in any action on the part of the resource addressed, but may be used to verify that a resource with the given address exists. If the status field of the reply header is nonzero then an error of InvalidAddress indicates that no resource with the supplied address exists.

C language syntax: None, but C language libraries may use the NOP action internally.

Coding:

action	sub-action	resource
3	21	0

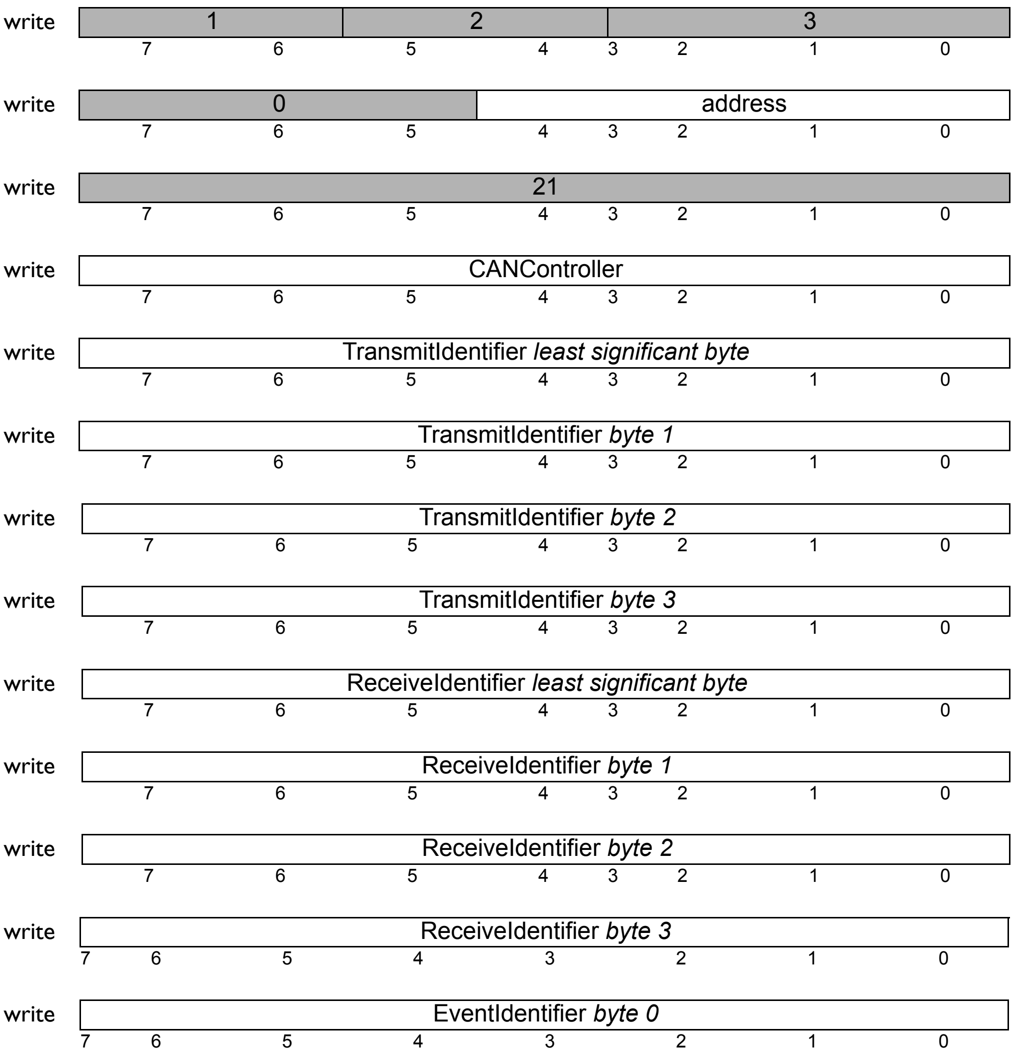
Arguments:

name	type	range
CANController	unsigned 8 bit	0
TransmitIdentifier	unsigned 32 bit	0-2047
ReceiveIdentifier	unsigned 32 bit	0-2047
EventIdentifier	unsigned 32 bit	0-2047

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	1-31

Packet Structure:


write
1
2
3
7 6 5 4 3 2 1 0
write
0
address
7 6 5 4 3 2 1 0
write
21
7 6 5 4 3 2 1 0
write
CANController
7 6 5 4 3 2 1 0
write
TransmitIdentifier *least significant byte*
7 6 5 4 3 2 1 0
write
TransmitIdentifier *byte 1*
7 6 5 4 3 2 1 0
write
TransmitIdentifier *byte 2*
7 6 5 4 3 2 1 0
write
TransmitIdentifier *byte 3*
7 6 5 4 3 2 1 0
write
ReceiveIdentifier *least significant byte*
7 6 5 4 3 2 1 0
write
ReceiveIdentifier *byte 1*
7 6 5 4 3 2 1 0
write
ReceiveIdentifier *byte 2*
7 6 5 4 3 2 1 0
write
ReceiveIdentifier *byte 3*
7 6 5 4 3 2 1 0
write
EventIdentifier *byte 0*
7 6 5 4 3 2 1 0

	7	6	5	4	3	2	1	0
write	EventIdentifier *byte 1*							

	7	6	5	4	3	2	1	0
write	EventIdentifier *byte 2*							

	7	6	5	4	3	2	1	0
write	EventIdentifier *byte 3*							

	7	6	5	4	3	2	1	0
read	2		status		*reserved*			

	7	6	5	4	3	2	1	0
read	PeriphAddress							

Description:

The **Open CAN Device** action is a request to a PRP device to return a PRP peripheral address associated with a CAN controller and two CAN identifiers on the device. ***CANController*** is the local physical CAN controller; for all current PRP devices there is at most one CAN controller, so this argument should be zero. ***TransmitIdentifier*** and ***ReceiveIdentifier*** are CAN identifiers used for sending and receiving messages. The point of view is the device, so **TransmitIdentifier** is used for sending messages from the PRP device to the peripheral CAN device, and ***ReceiveIdentifier*** should be used by the peripheral device to send messages to the PRP device. If either ***TransmitIdentifier*** or ***ReceiveIdentifier*** is zero than it will be ignored, and either transmit or receive disabled for the resulting peripheral.

EventIdentifer is a CAN identifier used for receiving asynchronous event notifications from a PRP device or Magellan Motion Processor. If the requested peripheral is associated with some other device then ***EventIdentifier*** should be zero.

The return value, ***PeriphAddress***, is a PRP address that may be used with the resource type ***Peripheral*** for addressing the newly opened CAN peripheral until it is closed.

C language interface:

```
PMDresult PMDPeriphOpenCAN(PMDPeriph *periph,
                           PMDDevice *device,
                           PMDuint32 TransmitIdentifier,
                           PMDuint32 ReceiveIdentifier,
                           PMDuint32 EventIdentifier);
```

OpenCMotionEngine Device

Coding:

action	sub-action	resource
3	3	0

Arguments:

name	type	range
CMEAddress	unsigned 8 bit	0

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	0			address				
write	3							
write	CMEAddress							
read	2		status		*reserved*			
read	RemoteAddress							

Description:

The **Open CMotionEngine Device** action is used to request a connection to a C-Motion Engine on a remote PRP device. The *CMEAddress* argument indicates which ***CMotionEngine*** resource on the remote device is to be used, for current PRP devices there is only one, so its address is always zero.

The returned *RemoteAddress* may be used as the address for, for example **CommandStartTask** actions to start a user program, **Send** and **Receive** actions to read and write user packets to a user program, and so forth.

It is not necessary to use **OpenCMotionEngine** to gain access to a C-Motion Engine on a local PRP device, that is, one that is directly connected to a host. For a local device one should simply use PRP address zero to address the C-Motion Engine.

C language syntax:

This call is performed as needed when opening a PRP device using the **PMDRPDeviceOpen** call. In the C interface separate handles to ***CMotionEngine*** resources are not required.

OpenISA Device

Coding:

action	sub-action	resource
3	19	0

Arguments:

name	type	range
ISAAddress	unsigned 16 bit	0-0xfff
EventIRQ	unsigned 8 bit	1-15

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	1-31

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	0			address				
write	19							
write	0							
write	ISAAddress *Byte 0*							
write	ISAAddress *Byte 1*							
write	*EventIRQ*							
read	2		status		*reserved*			
read	PeriphAddress							

Description:

The **Open ISA Device** action is a request to a Prodigy/CME device to return a PRP address for a peripheral for input and output to the ISA bus using the base address *ISAAdress*. The **Write** and **Read** actions may be used for output and input using addresses offset from the base address of the newly returned peripheral, or **Send** and **Receive** may be used for output and input at the base address.

EventIRQ is used to specify the interrupt channel used for signaling Magellan or Prodigy/CME asynchronous events. *EventIRQ* is not meaningful for peripherals that are not connected to a Magellan or Prodigy/CME device, and if not used should be set to zero.

C language syntax:

```
PMDresult PMDPeriphOpenISA(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint16 boardAddress,
                           PMDuint16 eventIRQ);
```

Open Memory32 Device

Coding:

action	sub-action	resource
3	2	0

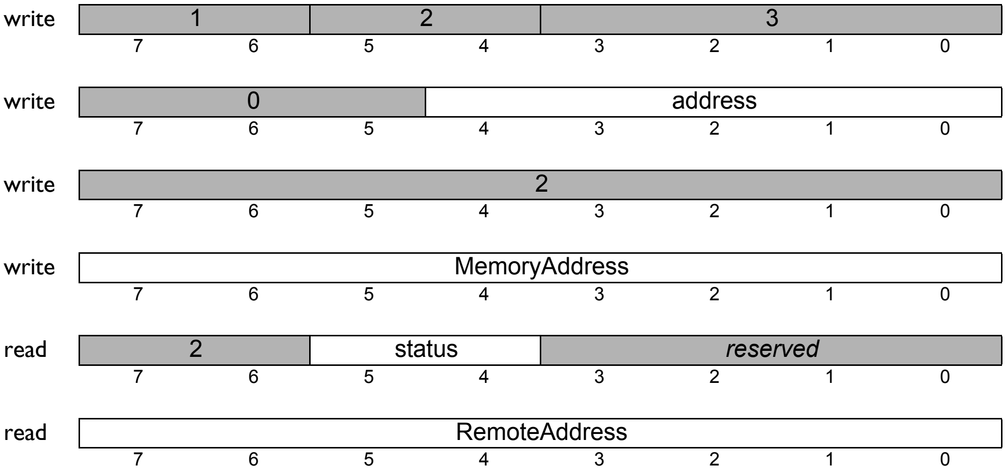
Arguments:

name	type	range
MemoryAddress	unsigned 8 bit	0-31

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:


write
1
2
3
7 6 5 4 3 2 1 0
write
0
address
7 6 5 4 3 2 1 0
write
2
7 6 5 4 3 2 1 0
write
MemoryAddress
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
RemoteAddress
7 6 5 4 3 2 1 0

Description:

The **Open Memory32 Device** action is used to request a connection to a ***Memory*** resource for 32-bit wide access on a remote PRP device. For current PRP devices the only ***Memory*** resource is the dual-ported RAM. The *MemoryAddress* argument indicates which ***Memory*** resource on the remote device is to be used, for current PRP devices there is only one, so its address is always zero.

The returned *RemoteAddress* may be used as the address when accessing the rersource, for example **Read** and **Write** actions to read and write values from a remote dual-ported RAM.

It is not necessary to use **Open Memory32** to gain access to a dual-ported RAM on a local PRP device, that is, one that is directly connected to a host. For a local device one may simply use PRP address zero to address the memory. **Open Memory32** will, however, return the correct address for a local device.

C language syntax:

```
PMDresult PMDMemoryOpen32(PMDMemoryHandle *hMemory,
                          PMDDeviceHandle *hDevice,
                          PMDMemoryId id);
```

Open MotionProcessor Device

Coding:

action	sub-action	resource
3	0	0

Arguments:

name	type	range
LocalAddress	unsigned 8 bit	0-31

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	0			address				
write	0							
write	LocalAddress							
read	2		status		*reserved*			
read	RemoteAddress							

Description:

The **Open MotionProcessor Device** action is used to request a connection to a Magellan Motion Processor that is part of a remote PRP device, that is, a device that is accessible only through another PRP device, and not directly via a TCP connection or other communication channel.

To access a motion processor on a local PRP device it is sufficient to use the local PRP address of the motion processor. Since all current PRP cards have one on-card motion processor that address is always zero.

LocalAddress is the local PRP address of the motion processor, as discussed above, this address is always zero for current PRP devices. The returned value *RemoteAddress* is a PRP address that may be used to send commands to the newly contacted motion processor. Once opened, the motion processor may be commanded in exactly the same way as a motion processor on a local device.

C language syntax:

```
PMDresult PMDAxisOpen(PMDAxisHandle *hAxis,
                      PMDDeviceHandle *hDevice,
                      PMDAxis axisNumber);
```

axisNumber is the motion processor axis to associate with the axis handle, *LocalAddress* in the C library case is always zero.

OpenCOM Device

Coding:

action	sub-action	resource
3	20	0

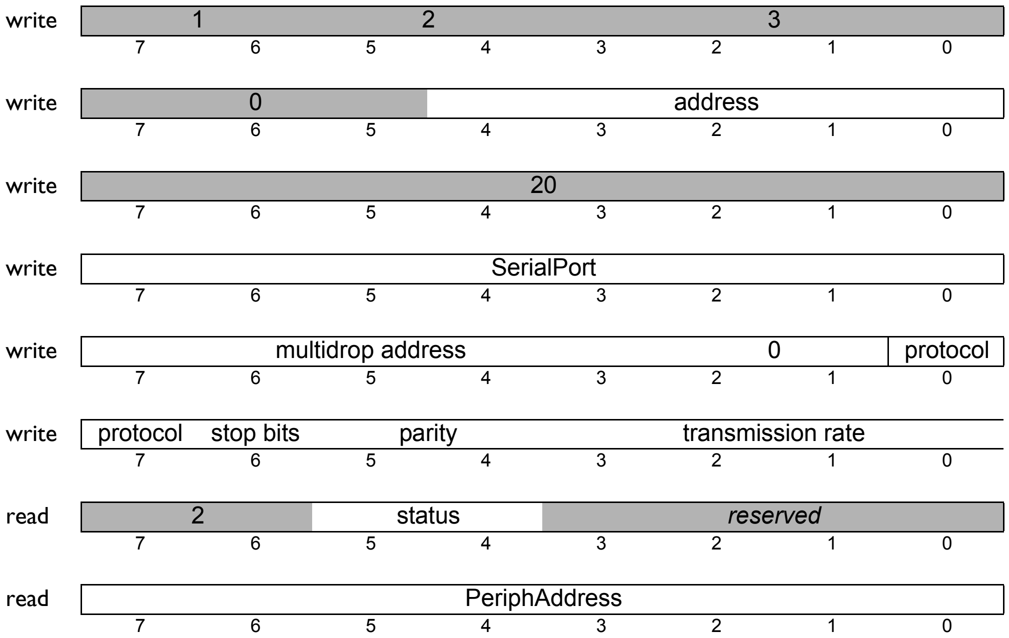
Arguments:

name	type	range
SerialPort	unsigned 8 bit	0-1
SerialMode	unsigned 16 bit	see below

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	1-31

Packet Structure:


write
1
2
3
7 6 5 4 3 2 1 0
write
0
address
7 6 5 4 3 2 1 0
write
20
7 6 5 4 3 2 1 0
write
SerialPort
7 6 5 4 3 2 1 0
write
multidrop address
0
protocol
7 6 5 4 3 2 1 0
write
protocol
stop bits
parity
transmission rate
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
PeriphAddress
7 6 5 4 3 2 1 0

Description:

The **Open COM Device** action is a request to a PRP device to return a PRP peripheral address associated with a serial port on the device. ***SerialPort*** is the local physical serial port on the device itself: 0 for COM1, and 1 for COM2. ***SerialMode*** is a 16 bit word encoding serial parameters as shown in the table below. The return value, ***PeriphAddress***, is a PRP address that may be used with the resource type **Peripheral** for addressing the newly opened serial peripheral until it is closed.

In order to open a peripheral that uses the PRP multi-drop serial protocol it is necessary to first open a COM peripheral using the **Open Device OpenCOM** action, and then to use the **Open Peripheral OpenMultiDrop** action.

SerialMode Encoding

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

SerialMode Encoding

Bit Number	Name	Instance	Encoding
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	3
9-10	reserved		0
10-15	multi-drop address	0	0
			-
		63	63

C language syntax:

```
PMDresult PMDPeriphOpenCOM(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDSerialPort port,
                           PMDSerialBaud baud,
                           PMDSerialParity parity,
                           PMDSerialStopBits stopbits);
```

OpenPAR Device

Coding

action	sub-action	resource
3	18	0

Arguments:

name	type	range
Address	unsigned 16 bit	0-0xffff
EventIRQ	unsigned 8 bit	0-0xff
MemoryWidth	unsigned 8 bit	1,2,4

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	0-0xff

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	0			address				
write	18							
write	Address *low byte*							
write	Address *high byte*							
write	EventIRQ							
write	MemoryWidth							
read	2		status		*reserved*			
read	RemoteAddress							

Description:

The **Open PAR Device** action is a request to open a connection to a parallel peripheral channel on a PRP device. Once such a peripheral is open the peripheral read or write actions may be used with it. *Address* is used to specify the channel to open; *MemoryWidth* to specify the size in bytes of data transfers, and *EventIRQ* to specify the interrupt in connection with the channel.

The return value *RemoteAddress* is a PRP address that may be used with resource type ***Peripheral*** for addressing the opened channel.

Currently only the ION/CME digital drive supports parallel peripherals, which are used for digital input/output and for analog input. Consult the *ION/CME Digital Drive User's Manual* for details.

C language interface:

```
PMDresult PMDPeriphOpenPAR(
                           PMDPeriphHandle*hPeriph
                           PMDDeviceHandle*hDevice,
                           WORD address,
                           BYTE EventIRQ,
                           PMDDataSize datasize);
```

OpenTCP Device

Coding:

action	sub-action	resource
3	22	0

Arguments:

name	type	range
EthernetInterface	unsigned 8 bit	0
IPAddress	unsigned 32 bit	0-0xffffffff
TCPPort	unsigned 16 bit	0-0xffff

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	1-31

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	0			address				
write	22							
write	EthernetInterface							
write	IPAddress *least significant byte*							
write	IPAddress *byte 1*							
write	IPAddress *byte 2*							
write	IPAddress *byte 3*							
write	TCPPort *byte 0*							
write	TCPPort *byte 1*							
read	2		status		*reserved*			
read	PeriphAddress							

Description:

The **Open TCP** action is a request to a PRP device to return a PRP peripheral address associated with an Ethernet TCP connection. *EthernetInterface* is the local physical Ethernet interface; for all current PRP devices there is one Ethernet interface, so this argument should be zero.

IPAddress is the remote address to which a connection should be opened. If *IPAddress* is zero, then the a port will be opened that will accept incoming connections, one incoming connection at a time may be handled by such a port. *TCPPort* is the TCP port to connect to or to listen on.

The return value, *PeriphAddress*, is a PRP address that may be used with the resource type ***Peripheral*** for addressing the newly opened Ethernet peripheral until it is closed.

C language interface:

```
PMDresult PMDPeriphOpenTCP(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint32 IPAddress,
                           PMDuint16 TCPPort);
```

OpenUDP Device

Coding:

action	sub-action	resource
3	23	0

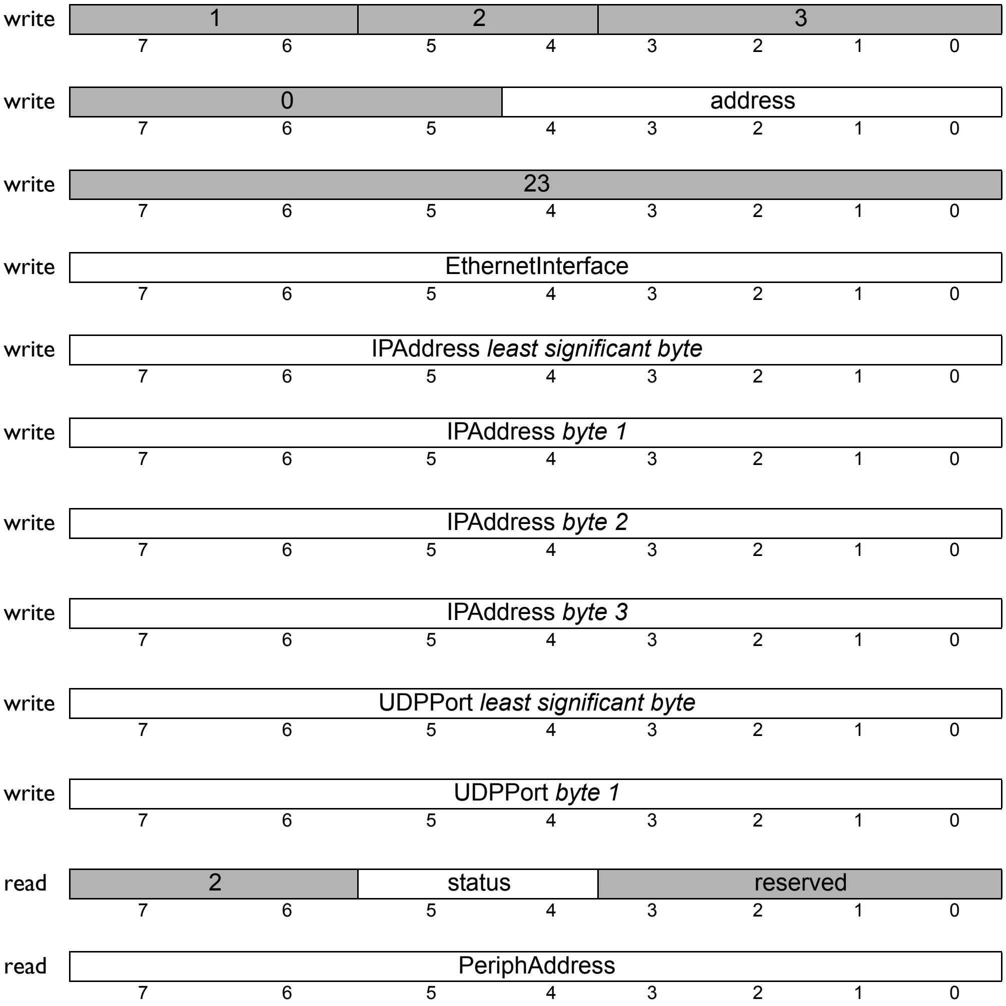
Arguments:

name	type	range
EthernetInterface	unsigned 8 bit	0
IPAddress	unsigned 32 bit	0-0xffffffff
UDPPort	unsigned 16 bit	0-0xffff

Returned Data:

name	type	range
PeriphAddress	unsigned 8 bit	1-31

Packet Structure:


write 1 2 3
write 0 address
write 23
write EthernetInterface
write IPAddress *least significant byte*
write IPAddress *byte 1*
write IPAddress *byte 2*
write IPAddress *byte 3*
write UDPPort *least significant byte*
write UDPPort *byte 1*
read 2 status reserved
read PeriphAddress
7 6 5 4 3 2 1 0

Description:

The **Open UDP Device** action is a request to a PRP device to return a PRP peripheral address associated with an Ethernet UDP port and remote IP address. *EthernetInterface* is the local physical Ethernet interface; for all current PRP devices there is one Ethernet interface, so this argument should be zero.

IPAddress is the remote address to which UDP packets should be sent. If *IPAddress* is zero then the a port will be opened that will accept incoming UDP packets. *UDPPort* is the UDP port to connect to or to listen on.

The return value, *PeriphAddress*, is a PRP address that may be used with the resource type ***Peripheral*** for addressing the newly opened Ethernet peripheral until it is closed.

C language interface:

```
PMDresult PMDPeriphOpenUDP(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint32 IPAddress,
                           PMDuint16 UDPPort);
```

Open Device Peripheral

Coding:

action	sub-action	resource
3	1	4

Arguments:

name	type	range
PeripheralAddress	unsigned 8 bit	0-31

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		3			
write	4			address				
write	1							
write	PeriphAddress							
read	2		status		*reserved*			
read	RemoteAddress							

Description:

The **Open Device Peripheral** action is used to allocate a PRP address for a ***Device*** resource that may be used to communicate with a PRP device accessible using an existing peripheral connection, for example a TCP or serial connection. The *RemoteAddress* returned may be used for any PRP action that may be addressed to a ***Device*** resource; it is typically used to obtain addresses for remote motion processors, dual-ported RAM, and C-Motion engines.

C language syntax:

```
PMDresult PMDRPDeviceOpen(PMDDeviceHandle *hDevice,
                          PMDPeriphHandle *hPeriph);
```

Open MotionProcessor Peripheral

Coding:

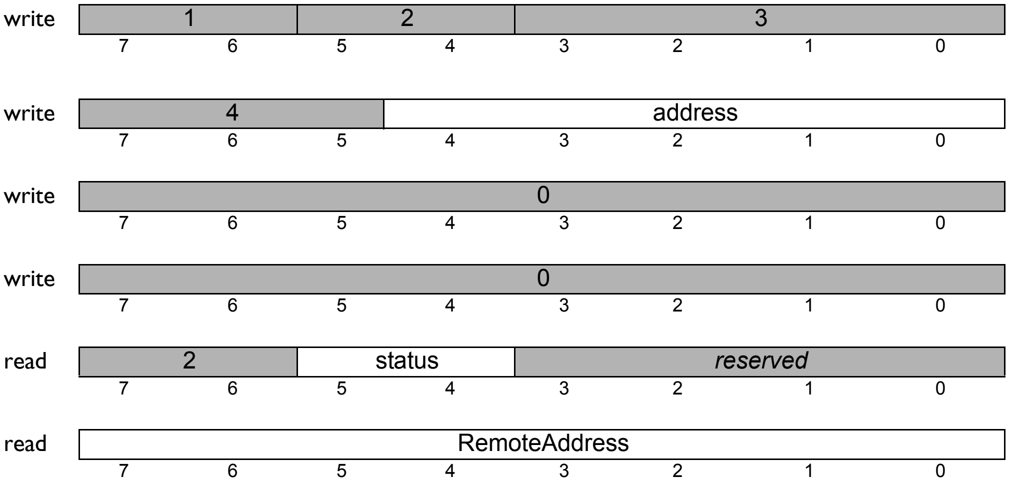
action	sub-action	resource
3	0	4

Arguments: none

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:


write
1
2
3
7 6 5 4 3 2 1 0
write
4
address
7 6 5 4 3 2 1 0
write
0
7 6 5 4 3 2 1 0
write
0
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
RemoteAddress
7 6 5 4 3 2 1 0

Description:

The **Open MotionProcessor Peripheral** action is used to allocate a PRP address to a Magellan Motion Processor that is accessible using an existing PRP peripheral resource, using a serial port, CAN bus, or PC-104 ISA bus. The PRP ***RemoteAddress*** returned may be used to command the motion processor using the **Command** action. The PRP device to which this action is directed will perform the translation from the PRP protocol for Magellan motion processor commands to the native Magellan protocol.

For example, to use a Prodigy/CME card to control an ION module on a CAN bus, one would:

1. Open a CAN peripheral with the CAN identifiers used by the module for command send and receive, using **OpenCAN** directed to the Prodigy/CME **Device.**
2. Use **Open MotionProcessor** to get an address for the remote ION using the peripherals opened in step 1.
3. Send commands to the remote ION using the **MotionProcessor** address returned in step 2.

C language syntax:

```
PMDresult PMDMPDeviceOpen(PMDDeviceHandle *hDevice,
                          PMDPeriph *hPeriph);
```

OpenMultiDrop Peripheral

Coding:

action	sub-action	resource
3	25	4

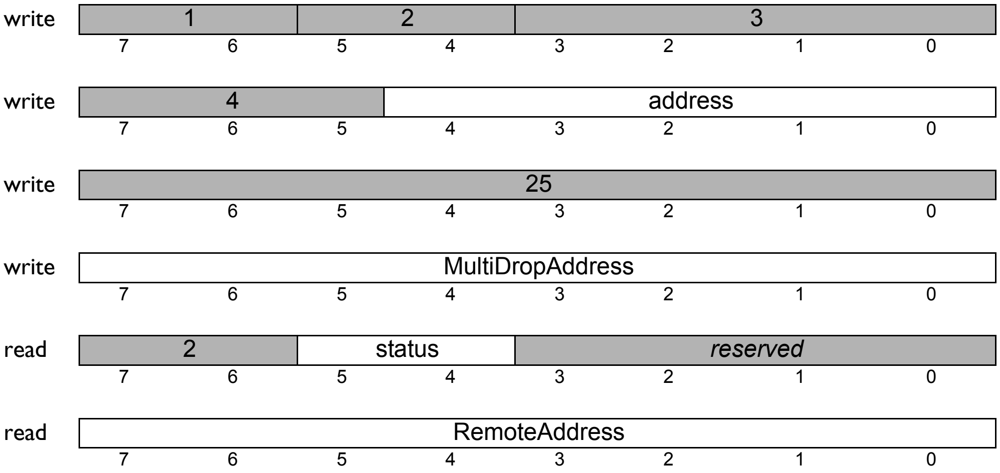
Arguments:

name	type	range
MultiDropAddress	unsigned 8 bit	0-31

Returned Data:

name	type	range
RemoteAddress	unsigned 8 bit	1-31

Packet Structure:


write
1
2
3
7 6 5 4 3 2 1 0
write
4
address
7 6 5 4 3 2 1 0
write
25
7 6 5 4 3 2 1 0
write
MultiDropAddress
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
RemoteAddress
7 6 5 4 3 2 1 0

Description:

The **Open MultiDrop Peripheral** action is used to obtain a peripheral that uses the PMD multi-drop serial protocol used for communicating with Magellan attached devices, such as non-CME ION modules, or with other PRP devices. The peripheral resource to which this action is directed must have been obtained using the **Open COM Device** action; the "parent" peripheral must not be closed before the multi-drop peripheral returned by **Open MultiDrop**, but should not be used for transmitting data on the serial line. The **RemoteAddress** returned by the **Open MultiDrop** action will typically be used as a target for **Open MotionProcessor** or **Open Device.**

For more information on the multi-drop protocol, see *Chapter 2, PMD Resource Access Protocol (PRP) Tutorial* and the *Magellan Motion Processor User's Guide.*

C language syntax:

```
PMDresult PMDPeriphOpenMultiDrop(PMDPeriphHandle *hPeriph,
                                 PMDPeriphHandle *hParent,
                                 unsigned MultiDropAddress);
```

Read Byte Peripheral

Coding:

action	sub-action	resource
8	1	4

Arguments:

name	type	range	units
Offset	unsigned 32 bit	0-0xffffffff	bytes
Length	unsigned 16 bit	0-0xffff	bytes

Returned Data: data bytes

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		8			
write	4			address				
write	1							
write	0							
write	Offset *byte 0*							
write	Offset *byte 1*							
write	Offset *byte 2*							
write	Offset *byte 3*							
write	Length *byte 0*							
write	Length *byte 1*							
read	2		status		*reserved*			
read	*data byte 0 …*							

Description:

The **Read Byte Peripheral** action is used to read a sequence of data bytes from a peripheral associated with a PC-104 ISA bus. The ***Offset*** argument is an offset from the base address that was specified when the peripheral was opened. The ***Length*** argument specifies the number of bytes to read; all bytes are read from the same addresses.

The data read is returned as the message body of the response packet.

This action is not applicable to other types of peripheral, and an InvalidResource error will be returned if another peripheral type is specified.

C language syntax:

```
PMDresult PMDPeriphRead(PMDPeriphHandle *hPeriph,
                        void *data,
                        PMDuint32 offset,
                        PMDuint32 length);
```

Read Dword Memory

Coding:

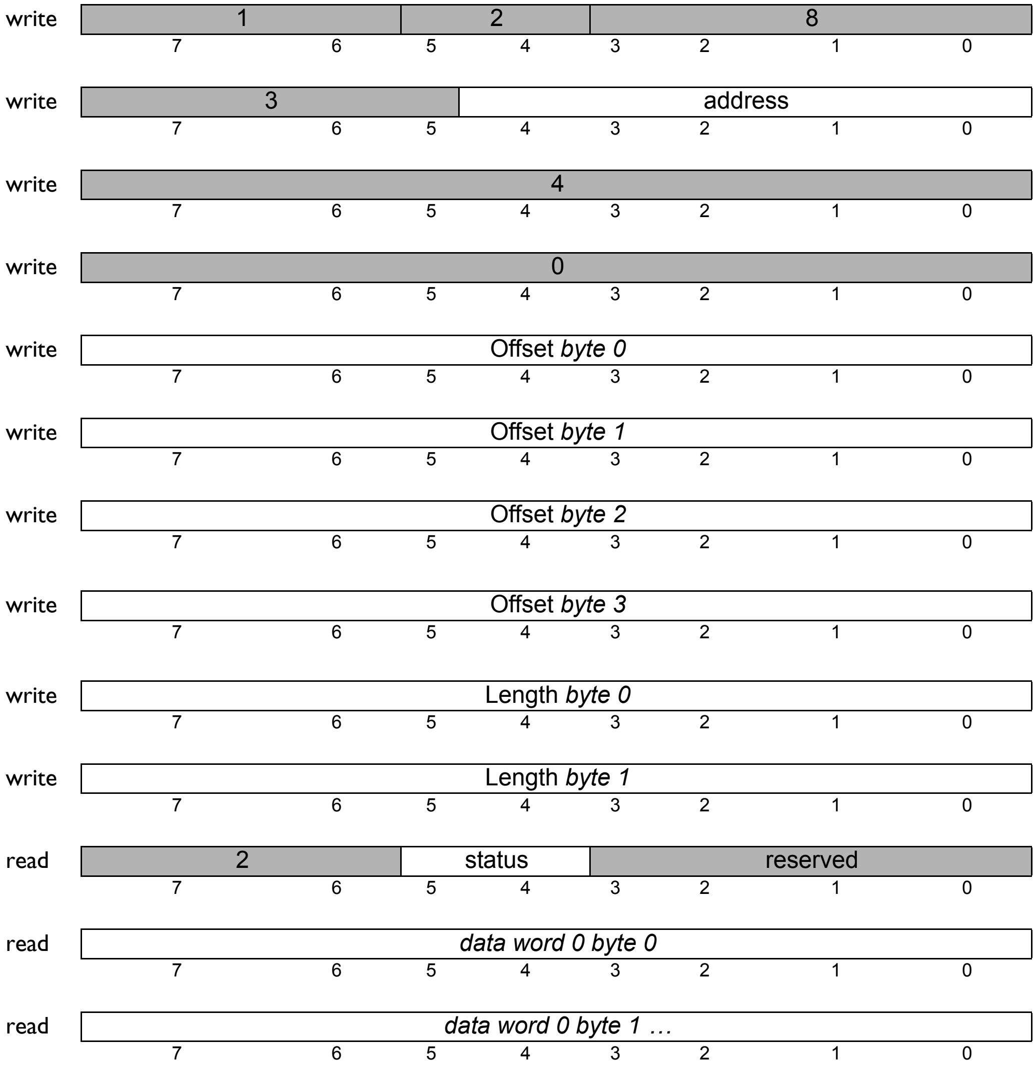
action	sub-action	resource
8	4	3

Arguments:

name	type	range	units
Offset	unsigned 32 bit	0-0xffffffff	bytes
Length	unsigned 16 bit	0-0xffff	double words

Returned Data: data bytes

Packet Structure:


write
1
2
8
write
3
address
write
4
write
0
write
Offset *byte 0*
write
Offset *byte 1*
write
Offset *byte 2*
write
Offset *byte 3*
write
Length *byte 0*
write
Length *byte 1*
read
2
status
reserved
read
data word 0 byte 0
read
data word 0 byte 1 …
7 6 5 4 3 2 1 0

Description: The **Read DWord Memory** action is used to read a sequence of 32 bit double words from a random access memory. The ***Offset*** argument is an address in the memory, typically an address in a dual-ported RAM. ***Offset*** should be divisble by four, the results of reading from a non-aligned address are unpredictable. The ***Length*** argument is the number of double words to read, exactly this number of double words are returned as the message body of the response packet.

C language syntax:

```
PMDresult PMDMemoryRead (PMDMemory *ram,
                         void *data,
                         PMDuint32 offset,
                         PMDuint32 length);
```

Read Word Peripheral

Coding:

action	sub-action	resource
8	1	4

Arguments:

name	type	range	units
Offset	unsigned 32 bit	0-0xffffffff	bytes
Length	unsigned 16 bit	0-0xffff	bytes

Returned Data: data bytes

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		8			
write	4			address				
write	1							
write	0							
write	Offset *byte 0*							
write	Offset *byte 1*							
write	Offset *byte 2*							
write	Offset *byte 3*							
write	Length *byte 0*							
write	Length *byte 1*							
read	2		status		*reserved*			
read	*data byte 0 …*							

Description:

The **Read Word Peripheral** action is used to read a sequence of 16 bit data words from a peripheral associated with a PC-104 ISA bus. The ***Offset*** argument is an offset from the base address that was specified when the peripheral was opened; ***Offset*** must be even. The *Length* argument specifies the number of bytes to read; ***Length*** must also be even. The data read is returned as the message body of the response packet.

The data read is returned as the message body of the response packet.

This action is not applicable to other types of peripheral, and an InvalidResource error will be returned if another peripheral type is specified.

C language syntax:

```
PMDresult PMDPeriphRead(PMDPeriphHandle *hPeriph,
                        void *data,
                        PMDuint32 offset,
                        PMDuint32 length);
```

Read Word IO

Coding:

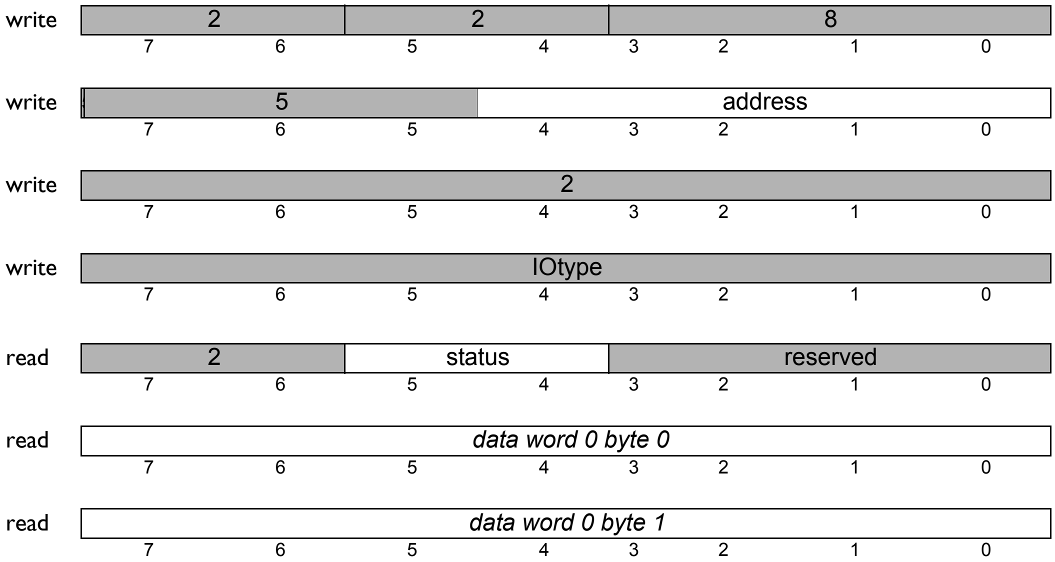
action	sub-action	resource
8	2	5

Arguments:

name	type	range	units
IOtype	unsigned 8 bit	01	PMDIOtype

Returned Data: data word

Packet Structure:


write
2
2
8
7 6 5 4 3 2 1 0
write
5
address
7 6 5 4 3 2 1 0
write
2
7 6 5 4 3 2 1 0
write
IOtype
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
data word 0 byte 0
7 6 5 4 3 2 1 0
read
data word 0 byte 1
7 6 5 4 3 2 1 0

Description: The **Read Word IO** action is used to read the value at an IO port of an ION/CME module.

C language syntax:

```
PMDresult PMDIORead (PMDIOHandle *hIO,
                     PMDuint16 *data
```

Receive CMotionEngine

Coding:

action	sub-action	resource
6	-	1

Arguments:

name	type	range	units
timeout	unsigned 16 bit	0-0xffff	msec
nExpected	unsigned 16 bit	0-0xffff	bytes

Returned Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		6			
write	1			address				
write	timeout *byte 0*							
write	timeout *byte 1*							
read	2		status		reserved			
read	*received data byte 0*							
read	*received data byte 1 …*							

Description:

The **Receive CMotionEngine** action is used to receive user packet data sent by a user program running on a C-Motion Engine. See the description of **Send CMotionEngine** (p. 68) for a description of the user packet mechanism. C-Motion user programs send user packets by calling **PMDPeriphSend** using a peripheral opened with the **PMDPeriphOpenCME** procedure.

The *timeout* argument specifies the maximum number of milliseconds to wait for data before failing with a PRP timeout error. A *timeout* value of 65535 (0xffff) means no time limit. In case of a timeout no bytes will be returned.

The C-Motion Engine buffers only one outgoing user packet at a time, so if no host is waiting to receive a user packet it may be overwritten by a newer user packet.

The size of the message received is given implicitly by the size of the return packet. How the size of the return packet is determined depends on the transport mechanism in use.

C language syntax:

```
PMDresult PMDPeriphOpenCME(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice);

PMDresult PMDPeriphReceive(PMDPeriphHandle *hPeriph,
                           void *buffer,
                           PMDuint32 *nReceived,
                           PMDuint32 nExpected,
                           PMDuint32 timeout);
```

Receive Peripheral

Coding:

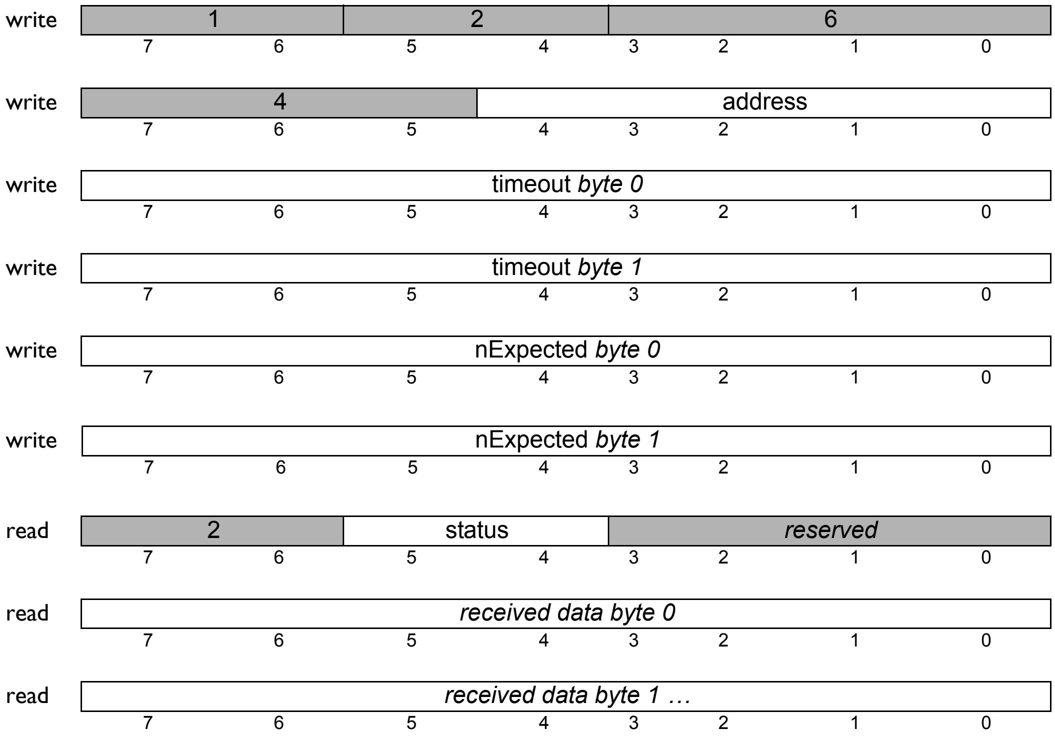
action	sub-action	resource
6	-	4

Arguments:

name	type	range	units
timeout	unsigned 16 bit	0-0xffff	msec
nExpected	unsigned 16 bit	0-0xffff	bytes

Returned Data: none

Packet Structure:


write
1
2
6
7 6 5 4 3 2 1 0
write
4
address
7 6 5 4 3 2 1 0
write
timeout byte 0
7 6 5 4 3 2 1 0
write
timeout byte 1
7 6 5 4 3 2 1 0
write
nExpected byte 0
7 6 5 4 3 2 1 0
write
nExpected byte 1
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0
read
received data byte 0
7 6 5 4 3 2 1 0
read
received data byte 1 …
7 6 5 4 3 2 1 0

Description:

The Receive Peripheral action is used to receive data from some remote device using the communication channel specified by the ***Peripheral*** resource to which it is addressed.

The *timeout* argument specifies the maximum number of milliseconds to wait for data before failing with a PRP timeout error. A *timeout* value of 65535 (0xffff) means no time limit. In case of a time out no bytes will be returned.

The *nExpected* argument specifies the maximum number of bytes to receive. For data that are naturally arranged in packets, for example TCP and UDP, only one packet will be received so the actual number of bytes returned may be less than *nExpected*. For data that are not arranged in packets, for example data received on a serial port peripheral, exactly *nExpected* bytes must be received or a timeout results and no data are returned.

The number of bytes of data actually returned is encoded in the size of the packet, how that size is transmitted depends on the transport mechanism.

If the peripheral connection has been closed by some external action, for example a TCP connection that has been closed by a peer, then a status of PMD_ERR_NotConnected will be returned. Such a peripheral must be closed using the Close action. In the case of a TCP connection, after closing the unconnected peripheral a new peripheral with the same TCP port may be opened using the OpenTCP action.

Receive Peripheral (cont.)

C language syntax:

```
PMDresult PMDPeriphReceive(PMDPeriphHandle *hPeriph,
                           void *buffer,
                           PMDuint32 *nReceived,
                           PMDuint32 nExpected,
                           PMDuint32 timeout);
```

Reset Device

Coding:

action	sub-action	resource
I	-	0

Arguments: none

Return Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	2		2		1			

	7	6	5	4	3	2	1	0
write	0			address				

Description: The **Reset Device** action may be used to soft reset a PRP device. No return packet will be sent after this command. The return packet for the next action will be a Reset error (0x8001) error reply, regardless of the action requested. A Reset error in reply to an action indicates that the command was not processed, and should be re-sent.

C language syntax:

```
PMDresult PMDDeviceReset(PMDDeviceHandle *hDevice);
```

Reset MotionProcessor

Coding:

action	sub-action	resource
1	-	2

Arguments: none

Return Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		1			
write	2			address				
read	2		status		*reserved*			

Description: The **Reset MotionProcessor** action may be used to hard reset a Magellan Motion Processor that is part of a PRP device. In order to soft reset a motion processor the **Command** action with a Magellan **reset** command may be used. It is an error to direct this action to a motion processor that is not part of a PRP device, for example an ION module.

C language syntax:

```
PMDresult PMDDeviceReset(PMDDeviceHandle *hDevice);
```

Send CMotionEngine

Coding:

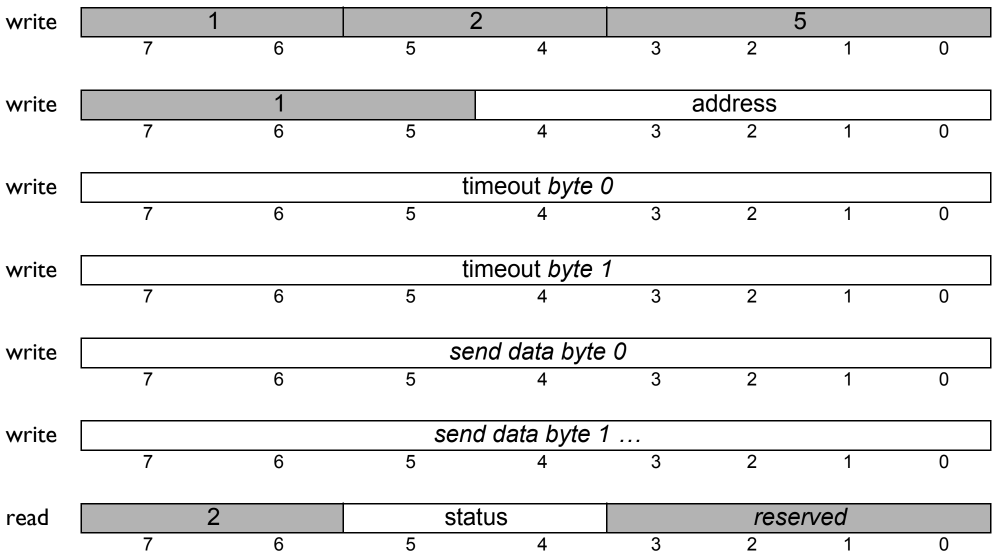
action	sub-action	resource
5	-	1

Arguments:

name	type	range	units
timeout	unsigned 16 bit	0-0xffff	msec

Returned Data: none

Packet Structure:


write
1
2
5
7 6 5 4 3 2 1 0
write
1
address
7 6 5 4 3 2 1 0
write
timeout byte 0
7 6 5 4 3 2 1 0
write
timeout byte 1
7 6 5 4 3 2 1 0
write
send data byte 0
7 6 5 4 3 2 1 0
write
send data byte 1 …
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0

Description:

The Send CMotionEngine action is used to send a user packet to a user program running on a C-Motion Engine, which may read them using the **PMDPeriphReceive** procedure applied to a peripheral opened with **PMDPeriphOpenCME**. The user packet mechanism allows arbitrary user data to be sent to or received from user programs without opening dedicated peripheral channels – the packets are encapsulated in PRP packets. User packets are sent as discrete units, and only one packet may be buffered before being read by a user program.

The ***timeout*** argument specifies how many milliseconds to wait for the user program to read the user packet. A ***timeout*** value of 65535 (0xffff) means no time limit.

The user packet mechanism is the simplest way to exchange data with running C-Motion Engine user programs, and has the advantage of working the same way regardless of the transport mechanism used to send packets, but it is limited in performance and flexibility. If user packets are not sufficient then peripheral channels specific to the user application should be opened and used.

The maximum size of a user packet is 250 bytes, as given by USER_PACKET in the file PMDPeriph.h. The actual size of the user packet sent is implicitly given by the size of the outgoing PRP packet. How the PRP packet size is determined depends on the transport mechanism in use.

C language syntax:

```
PMDresult PMDPeriphOpenCME(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice);

PMDresult PMDPeriphSend(PMDPeriphHandle *hPeriph,
                        void *buffer,
                        PMDuint32 nCount,
                        PMDuint32 timeout);
```

Send Peripheral

Coding:

action	sub-action	resource
5	-	5

Arguments:

name	type	range	units
timeout	unsigned 16 bit	0-0xffff	msec

Returned Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		5			
write	4				address			
write	timeout *byte 0*							
write	timeout *byte 1*							
write	*send data byte 0*							
write	*send data byte 1 …*							
read	2		status		*reserved*			

Description:

The **Send Peripheral** action is used to transmit data to some remote device using the communication channel specified by the ***Peripheral*** resource to which it is addressed. The peripheral might be a TCP Ethernet connection, a serial port, pair of CAN bus identifiers, or any other peripheral type. The number of bytes to send is implicit in the size of the PRP packet, how this is determined depends on the transport mechanism in use.

If all of the data cannot be sent within ***timeout*** milliseconds then a PRP timeout error will be returned. In which case some of the data may have been sent, it is not possible to tell. A ***timeout*** value of 65535 (0xffff) means no time limit.

If the peripheral connection has been closed by some external action, for example a TCP connection that has been closed by a peer, then a status of PMD_ERR_NotConnected will be returned. Such a peripheral must be closed using the **Close** action. In the case of a TCP connection, after closing the unconnected peripheral a new peripheral with the same TCP port may be opened using the **OpenTCP** action.

C language syntax:

```
PMDresult PMDPeriphSend(PMDPeriphHandle *hPeriph,
                        void *buffer,
                        PMDuint32 nCount,
                        PMDuint32 timeout);
```

Set Console CMotionEngine

Coding:

action	sub-action	resource
9	4	3

Arguments:

name	type	meaning
Console	unsigned 8 bit	peripheral address

Returned Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	1		2		9			
write	3			address				
write	4							
write	Console							
read	2		status		*reserved*			

Description:

The **Set Console CMotionEngine** action is used to change the destination of console messages from a user program running in the C-Motion Engine to which the action is addressed. User programs can emit console messages using the C library procedure PMDprintf. Console messages are primarily intended for debugging and routine progress monitoring.

The ***Console*** argument is the address of a peripheral to be used for console output. If **Console** is zero, then all console output will be suppressed. If ***Console*** is nonzero it must be the address of a peripheral that was opened on the same device as the C-Motion engine being addressed – if it is an inappropriate peripheral address then an error will be returned.

C language syntax:

None, this action is not currently supported by the C library. Pro-Motion may be used to change the current console.

SetDefault Device

Coding:

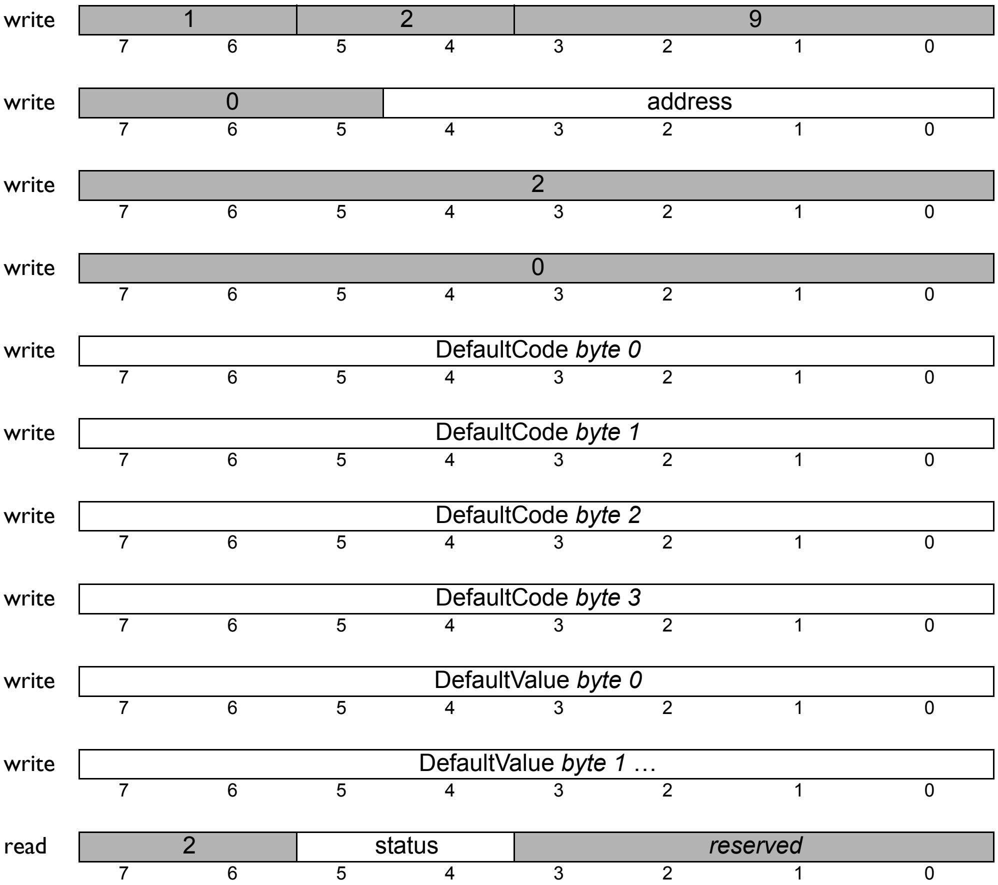
action	sub-action	resource
9	2	0

Arguments:

name	type	meaning
DefaultCode	unsigned 32 bit	default identifier
DefaultValue	varies	

Returned Data: none

Packet Structure:


write
1
2
9
7 6 5 4 3 2 1 0
write
0
address
7 6 5 4 3 2 1 0
write
2
7 6 5 4 3 2 1 0
write
0
7 6 5 4 3 2 1 0
write
DefaultCode *byte 0*
7 6 5 4 3 2 1 0
write
DefaultCode *byte 1*
7 6 5 4 3 2 1 0
write
DefaultCode *byte 2*
7 6 5 4 3 2 1 0
write
DefaultCode *byte 3*
7 6 5 4 3 2 1 0
write
DefaultValue *byte 0*
7 6 5 4 3 2 1 0
write
DefaultValue *byte 1 …*
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0

Description:

The **Set Default Device** action is used to change various non-volatile properties of a PRP device, for example the IP address, or whether to run a user program immediately after power up. The length of ***DefaultValue*** depends on the particular data type, and is encoded in the upper byte of ***DefaultCode***. The length in bytes is the field value minus one; a length value of zero means one byte, one means two bytes. Most default values are either two or four bytes long, but some are longer.

The table below summarizes the set of default values and their codes:

Prodigy/CME Defaults

name	code	length (bytes)	factory default
DefaultCPMotorType	0x0102	2	0x7777 (All axes set to brushed)
DefaultIPAddress	0x0303	4	0xC0A80202 (192.168.2.2)
DefaultNetMask	0x0304	4	0xFFFFFF00 (255.255.255.0)

Prodigy/CME Defaults

name	code	length (bytes)	factory default
DefaultGateway	0x0305	4	0x00000000 (0.0.0.0)
DefaultTCPPort	0x0106	2	40100
DefaultCOM1Mode	0x010E	2	0x0004 (57600,n,8,1)
DefaultCOM2Mode	0x010F	2	0x0005 (115200,n,8,1)
DefaultRS485Duplex	0x0110	2	0 (Full duplex)
DefaultCANMode	0x0111	2	0x0000 (1000 kbs)
DefaultAutoStartMode	0x0114	2	0
DefaultConsoleIntfType	0x0118	2	4 (Serial)
DefaultConsoleIntfAddr	0x0119	2	1 (PMDSerialPort2)
DefaultConsoleIntfPort	0x011A	2	5 (PMDSerialBaud115200)

All other values reserved.

DefaultIPAddress is the IP address of the Ethernet controller. It is typically necessary to set this default using the serial interface to suit the network in which a PRP device is to be installed. The default value is chosen to be part of a reserved IP class, and is not routable on the Internet.

Note that IP addresses are typically written in "dotted quad" notation, where each byte is written in decimal, separated by a dot. In order to convert from dotted quad notation to hexadecimal write convert each dot-separated field to hexadecimal and concatenate.

DefaultNetMask is a bitmask defining which IP addresses are directly accessible in the local subnet, the default is for a class C network, and must typically be changed to suit the network in which the PRP device is installed.

DefaultGateway is the IP address of the router to be used for all non-local IP addresses. PRP devices does not support more general routing tables because it is expected that they will usually deal with hosts on the local network. *DefaultGateway* must be changed to enable routing to any non-local IP addresses, but that such routing may not be necessary for many applications.

DefaultTCPPort is the base TCP port used for accepting host commands. In most cases there is no reason to change the default value of 40100.

DefaultCOM1Mode and *DefaultCOM2Mode* are serial port modes with the same meaning as **SerialMode** in the **OpenSerial** action, and are applied to the two serial ports immediately after coming out of reset. Serial port modes may be changed later by using the **OpenSerial** action.

DefaultRS485Duplex controls whether duplex mode is used in case serial port COM1 is configured as for RS-485. One means full-duplex, zero means half-duplex.

DefaultCANMode is an encoding of CAN bus parameters similar to that used by Magellan, as described in the *Magellan Motion Processor Programmer's Command Reference*, and are summarized below. The CAN mode cannot be changed except by using *DefaultCANMode*, it cannot be changed "on the fly."

DefaultCANMode fields

Bits	Name	Instance	Encoding
0-6	CAN NodeID	Node0	0
		Node1 …	1
		Node127	127
7-12	reserved		0

DefaultCANMode fields

Bits	Name	Instance	Encoding
13-15	Transmission Rate	1,000,000 baud	0
		800,000 baud	1
		500,000 baud	2
		250,000 baud	3
		125,000 baud	4
		50,000 baud	5
		20,000 baud	6
		10,000 baud	7

All CAN devices on the same bus must use the same transmission rate in order to communicate properly. The *CAN NodeID* encodes a set of CAN identifiers to be used for accepting host commands and returning responses, and uses the same scheme as do Magellan Motion Processors. All PRP devices and all Magellan Motion Processors on the same CAN bus must have distinct NodeIDs. Messages with a CAN identifier of 0x600 + NodeID will be accepted as PRP host commands, and will be responded to using CAN identifier 0x580 + NodeID. Asynchronous event notification messages will be sent using CAN identifier 0x180 + NodeID.

DefaultAutoStartMode controls whether a user program in the C-Motion Engine will be run automatically after coming out of reset. A value of one means that any user program present will be automatically run, zero means that a user program will not be run until a **CommandTaskStart** action is received. Automatic starting of user programs will be inhibited if a user program has caused a previous reset, for example by causing an exception.

DefaultConsoleIntfType, *DefaultConsoleIntfAddr*, and *DefaultConsoleIntfPort* determine the communications channel that will be used for console (user program output) messages. The channel used may be changed at run time by using the **Set ValueConsole** action. The encoding of these default values is explained in the table below.

Console Output Defaults

DefaultConsoleIntfType value	peripheral type	DefaultConsoleIntfAddr meaning	DefaultConsoleIntfPort meaning
0	none	ignored	ignored
1		*reserved*	
2	PCI	ignored	ignored
3		*reserved*	
4	serial	0 – COM1, 1 – COM2	port settings
5		*reserved*	
6		*reserved*	
7	UDP	IP address	UDP port
> 7		*reserved*	

C language syntax:

```
PMDresult PMDSetDefault(PMDDeviceHandle *hDevice,
                        PMDDefault default,
                        void *value,
                        unsigned valueSize);
```

WriteDWord Memory

Coding:

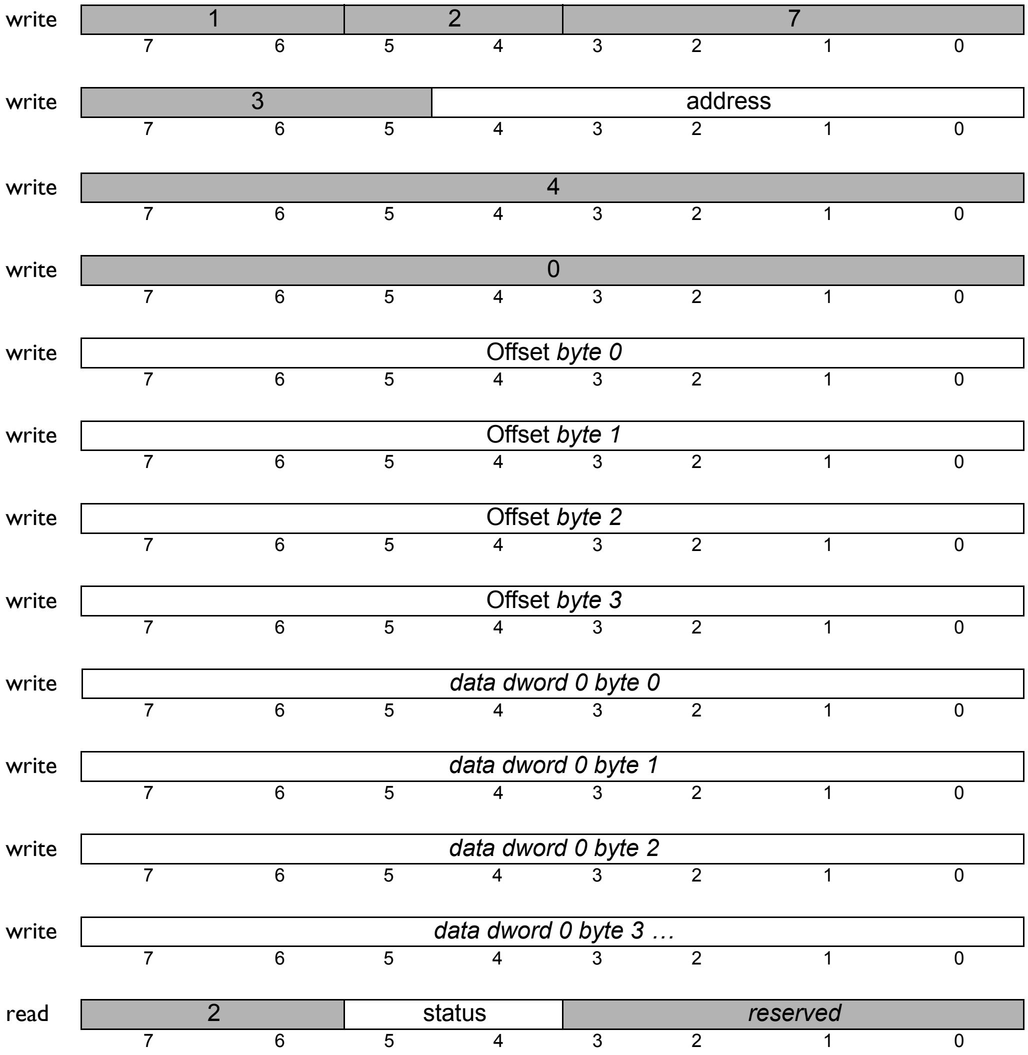
action	sub-action	resource
7	4	3

Arguments:

name	type	range	units
Offset	unsigned 32 bit	0-0xffffffff	bytes

Returned Data: none

Packet Structure:


write
1
2
7
write
3
address
write
4
write
0
write
Offset byte 0
write
Offset byte 1
write
Offset byte 2
write
Offset byte 3
write
data dword 0 byte 0
write
data dword 0 byte 1
write
data dword 0 byte 2
write
data dword 0 byte 3 …
read
2
status
reserved
7
6
5
4
3
2
1
0

Description:

The **Write DWord Me‘mory** action is used to write a sequence of four byte (32 bit) double words to a random access memory. The *Offset* argument is an index or address into the memory, typically an address in a dual-ported RAM. *Offset* should be divisible by four, the result of a non-aligned write is not predictable. As many double words as are supplied in the packet are written to memory, if the number of bytes supplied is not divisible by four the results are unpredictable.

C language syntax:

```
PMDresult PMDMemoryWrite(PMDMemoryHandle *hRam,
                         void *data,
                         PMDuint32 offset,
                         PMDuint32 length);
```

Coding:

WriteByte Peripheral

Coding:

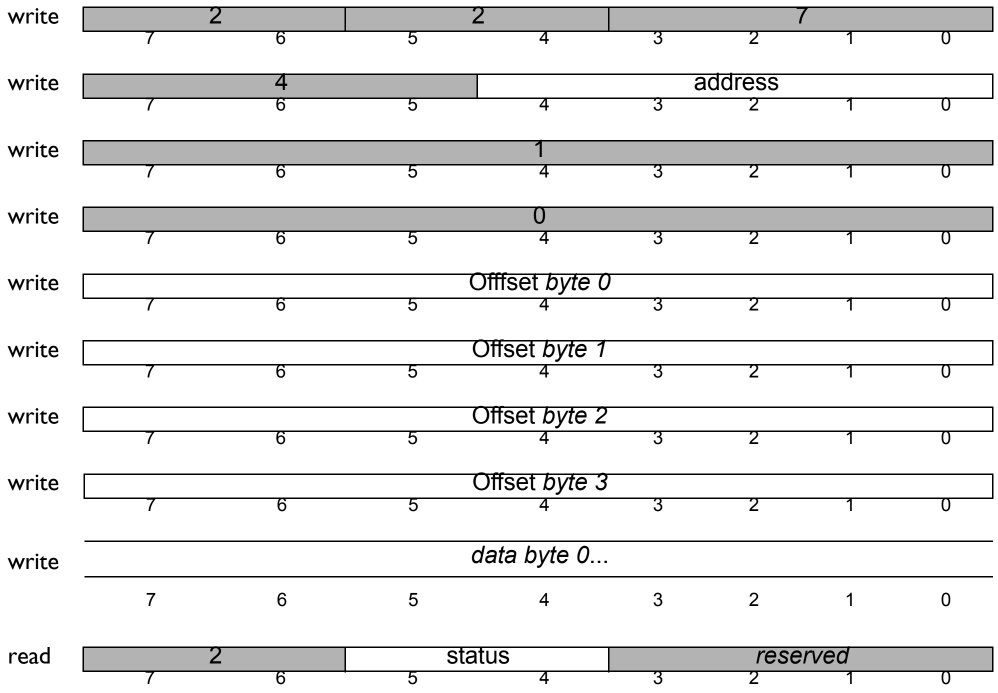
action	sub-action	resource
7	1	4

Arguments:

name	type	range	units
Offset	unsigned 32 bit	0-0xffffffff	bytes

Returned Data: none

Packet Structure:


write
2
2
7
7 6 5 4 3 2 1 0
write
4
address
7 6 5 4 3 2 1 0
write
1
7 6 5 4 3 2 1 0
write
0
7 6 5 4 3 2 1 0
write
Offfset *byte 0*
7 6 5 4 3 2 1 0
write
Offset *byte 1*
7 6 5 4 3 2 1 0
write
Offset *byte 2*
7 6 5 4 3 2 1 0
write
Offset *byte 3*
7 6 5 4 3 2 1 0
write
data byte 0...
7 6 5 4 3 2 1 0
read
2
status
reserved
7 6 5 4 3 2 1 0

Description:

The **Write Byte Peripheral** action is used to write a sequence of data bytes to a peripheral associated with a PC-104 ISA bus. The ***Offset*** argument is an offset from the base address that was specified when the peripheral was opened. As many bytes as are supplied in the packet are written to the ISA bus from the address given by the base address plus ***Offset***.

This action is not applicable to other types of peripheral, and an InvalidResource error will be returned if another peripheral type is specified.

C language syntax:

```
PMDresult PMDPeriphWrite (PMDPeriphHandle *hPeriph,
                          void *data,
                          PMDuint32 address,
                          PMDuint32 length);
```

Write Word IO

Coding:

action	sub-action	resource
7	2	5

Arguments:

name	type	range	units
IOtype	unsigned 8 bit	0-1	PMDIOtype
Mask	unsigned 16 bit	0-0xffff	bit mask
	unsigned 16 bit	0-0xffff	IO dependent

Returned Data: none

Packet Structure:

	7	6	5	4	3	2	1	0
write	2		2		7			
write	5			address				
write	2							
write	IOtype							
write	Mask *byte 0*							
write	Mask *byte 1*							
write	data word 0 *byte 0*							
write	data word 0 *byte 1*							
read	2		status		*reserved*			

Description:

The **Write Word IO** action is used to write a value to a general purpose IO port (digital or analog) on an ION/CME module. If the IO port type being addressed is a digital IO port, the mask field is used to specify which bits will be affected by the write.

C language syntax:

```
PMDresult PMDIOWrite (PMDIOHandle *hIO,
PMDuint16 *data,
PMDuint16 mask)
```

4. PMD C Language Library Procedures

PRP device products include C language libraries for control and communication. Library versions are provided for host PCs and for the C-Motion Engine included in Prodigy/CME and ION CME. TheCME libraries provide the following features:

- Resource virtualization. Local and remote resources, for example, motion processors, Prodigy/CME cards and ION/CME modules are controlled in the same way.
- Procedure call syntax is the same for host programs as for C-Motion Engine programs. (There are a few procedures only applicable to one environment).
- Host libraries can be easily linked with any C or C++ program, sources are provided.
- C-Motion Engine programs can use standard C libraries.

The PMD C library for controlling Magellan Motion Processors is called “C-Motion,” and is fully documented in the *Magellan Motion Processor Programmer’s Command Reference* and the *Magellan Motion Processor User’s Guide*.

4.1 Naming Conventions

Procedures and data type names in the CME library are prefixed with “PMD.” This prefix is omitted in the binary protocol documentation below, but must be included in C programs. *C-Motion* is the PMD library for Magellan Motion Processor control, and is a subset of the CME libraries. C-Motion procedures and data type names are also prefixed with “PMD.”

4.2 Data Types

PRP resources are represented by opaque C types. “Opaque” means that reading and writing members of the data structures without using the library procedures is not supported. All of these structures must be allocated by the calling program, are passed to library procedures by using a pointer argument, and must not be freed or otherwise written to until explicitly closed.

These data types include:

- **PMDDeviceHandle** – There are two types of “device:” an *RP device* is a device that communicates using the PRP, that is, a Prodigy/CME card or an ION/CME module; an MP device is a device that communicates using the Magellan protocol, that is, a non-CME ION module, non-CME prodigy card, or other “Magellan attached” device.
- **PMDAxisHandle** – A control axis of a Magellan Motion Processor, which may be part of a Magellan attached device or of a PRP device.
- **PMDPeriphHandle** – A connection to a peripheral device over a particular communication channel. The peripheral data type specifies both the communication channel and any addressing information specific to a remote device, for example a TCP/IP port number or a PC-104 ISA bus base address.
- **PMDMemoryHandle** – A dual-ported RAM on a PRP device or a non-CME Prodigy card.

The include file “PMDtypes.h” defines typedefs for specific integral types that will be used in the prototypes in this manual:

- PMDuint32, PMDint32 – unsigned and signed 32-bit integers
- PMDuint16, PMDint16 – unsigned and signed 16-bit integers
- PMDuint8, PMDint8 – unsigned and signed 8-bit integers

Many bitmask and enumerated types are also defined in this file.

4.3 Return Values

Almost all of the PMD library procedures return an integer of type **PMDresult**, indicating success (zero) or failure (nonzero). The error values of **PMDresult** are the same as the PRP error values documented in this manual, and are all declared in the “PMDecode.h” file. A partial list of these error codes is in for more information.

4.4 C-Motion Engine Macros

A number of C preprocessor macros are required as part of a C-Motion Engine user code program. These macros are defined in the “UserMain.h” file.

```
USER_CODE_VERSION (MAJOR, MINOR)
USER_CODE_TASK ( UserTCP )
```

USER_CODE_VERSION encodes version information in a section of the binary that will be used by the C-Motion Engine runtime code. It should be put once in the main source file at top level (outside of any function definition).

MAJOR and MINOR are user program version numbers, 16-bit constants that will be reported by Pro-Motion. USER_CODE_VERSION must be present even if you don’t care to maintain a version number.

USER_CODE_TASK should be used to define the main function of the user code program, its argument is the name of the function, which should accept no arguments and should never return. A user program skeleton follows:

```
#include “c-motion.h”
#include "usermain.h"
#include "PMDsys.h"

// this macro is required at the beginning of a CME user application
USER_CODE_VERSION (1,0)
// UserTCP is the name of the main task function
USER_CODE_TASK (UserTCP)
{
…

 while (l) {
 // Handle task events
}
// Note: do not return
}
```

4.5 PMD Library Procedures

This section documents the PMD C language interface to the library procedures for programming aCME PRP device, both in hosted programs and C-Motion Engine user programs. Most procedure calls are syntactically the same in both environments, but their implementation is of course quite different.

In many cases a PRP action corresponds closely to the action of a library procedure, but this is not invariable. One procedure call may involve multiple PRP actions, or none, and whether PRP is used may depend on whether the procedure call is executed on the host or in a C-Motion Engine user program, and on whether it is directed at a remote device or the device on which the program itself is running.

There are a few conventions common to many procedures:

- When opening a handle to some object a pointer to an uninitialized instance of the appropriate data type is passed first, and the open procedure will write to it. The initialized data type should not be written to as long as it is in use.
- Most procedures return an integer status code of type PMDresult. A zero indicates success, and a non-zero value failure or error.
- Many procedures that accept a pointer to a PMDDeviceHandle as an argument should be passed a null pointer to indicate the "local" device. For C-Motion Engine user programs the local device is the device hosting the C-Motion Engine. For hosted programs, for example when opening a peripheral, the local device is the host itself.

This page intentionally left blank.

PMDAxisOpen

C-Motion Engine	Host-Based

Arguments:

name	type
hAxis	pointer to PMDAxisHandle
hDevice	pointer to PMDDeviceHandle
axis_number	enumeration PMDAxis1 to PMDAxis4

C language syntax:

```
PMDresult PMDAxisOpen(PMDAxisHandle *hAxis,
                      PMDDeviceHandle *hDevice,
                      PMDAxis axis_number);
```

Visual Basic Syntax:

```
Dim axis As New(device, PMDAxisNumber.Axis)
```

Description:

PMDAxisOpen is used to obtain a handle to a single control axis of a Magellan Motion Processor, which will be used for all C-Motion procedures. The ***hAxis*** argument should point to an uninitialized **PMDAxisHandle** struct, which should not be freed or written to as long as the handle is required. The ***device*** argument should point to an open **PMDDeviceHandle** handle, which may represent either a PMD device or a Magellan attached device. In a C-Motion engine user program, **device** may be null, in which case the Magellan processor on the local device will be opened.

For example, to open the first axis on the local Magellan processor from a CME user program:

```
PMDAxisHandle axis1;
PMDresult result;

result = PMDAxisOpen(&axis1, 0, PMDAxis1);
```

And to open the second axis on a Magellan attached device accessible by CANBus:

```
PMDPeriphHandle periph;
PMDDeviceHandle dev;
PMDAxisHandle axis2;
PMDresult result;

// First open the peripheral connection, CAN_TX, CAN_RX, and CAN_EVENT
//  depend on how the attached device is configured.
result = PMDPeriphOpenCAN(&periph, 0, CAN_TX, CAN_RX,
                                                  CAN_EVENT);
// Now open an MP Device on the peripheral
if (PMD_NOERROR == result)
  status = PMDMPDeviceOpen(&dev, &periph);
// Now we’re ready to obtain the axis handle.
if (PMD_NOERROR == result)
  result = PMDAxisOpen(&axis2, &dev, PMDAxis2);
```

Related PRP Actions:

Open Peripheral MotionProcessor

PMDTaskGetState

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle
state	pointer to PMDTaskState enum

C language syntax:

```
PMDresult PMDTaskGetState(PMDDeviceHandle *hDevice,
                          PMDTaskState *state);
```

Visual Basic Syntax:

```
Dim state As PMDTaskState
state = device.TaskStat
```

Description:

The **PMDTaskGetState** procedure queries a C-Motion Engine for the state of any user program that might be installed in it. The *hDevice* argument should be associated with an RP device that is a device containing a C-Motion Engine. If **hDevice** is not appropriate then **PMD_ERR_NOT_SUPPORTED** will be returned.

The value pointed to by the state argument will be written to indicate the result:

PMDTaskState instance	encoding
No program installed	1
Program not started	2
Program running	3
Program aborted	4

Related PRP Actions:

Get CMotionEngine TaskState

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle

C language syntax:

```
PMDresult PMDTaskStart(PMDDeviceHandle *hDevice);
```

Visual Basic Syntax:

```
device.TaskStart()
```

Description:

PMDTaskStart is used to start a user program installed in the C-Motion Engine that is part of the CME device associated with the ***hDevice*** argument. If **hDevice** is not a PRP device then **PMD_ERR_Not_Supported** will be returned. If no runnable program is installed then **PMD_ERR_UC_NotProgrammed** will be returned. If a program is already running, then **PMD_ERR_UC_TaskAlreadyRunning** will be returned.

Related PRP Actions:

Command CMotionEngine Task

PMDTaskStop

Arguments:

name	**type**
hDevice	pointer to PMDDeviceHandle

C language syntax:

```
PMDresult PMDTaskStop(PMDDeviceHandle *hDevice);
```

Visual Basic Syntax:

```
device.TaskStop()
```

Description:

PMDTaskStop is used to stop any user program currently running in the C-Motion Engine that is part of the PRP device associated with the ***hDevice*** argument. If **device** is not a CME PRP device then **PMD_ERR_NOT_SUPPORTED** will be returned. If no program is currently running, then PMD_ERR_UC_TaskNotCreated will be returned. If no program is installed, then PMD_ERR_UC_NotProgrammed will be returned.

It is the user's responsibility to ensure safety when starting or stopping a user program that controls motors. It is not possible to predict the state of the PRP device or of it's motion processor at the instant that the user program is stopped. PMD strongly recommends that a task be stopped only to correct unrecoverable errors and that the card and any devices that it controls be put immediately into a known safe state using host commands. Because the card resources and the dynamic heap are not in a known state it is not safe to restart a task after stopping it without first resetting the entire device.

Related PRP Actions:

Command CMotionEngine CommandTask TaskStop

PMDDeviceClose

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle

C language syntax:

```
PMDresult PMDDeviceClose(PMDDeviceHandle *hDevice);
```

Visual Basic Syntax:

```
device.Close()
```

Description: **PMDDeviceClose** is used to free any resources used in maintaining the device handle passed as a pointer argument. After closing the memory used for the **PMDDeviceHandle** type may be freed or re-used for another device.

Related PRP Actions:

Close Device

Close CMotionEngine

PMDDeviceGetDefault

Arguments:

name	type
hDevice	pointer to open RP device handle
defaultcode	enumerated default code
value	pointer to memory area to receive default value
valueSize	maximum size of value area

C language syntax:

```
PMDresult PMDDeviceGetDefault(PMDDeviceHandle *hDevice,
                              PMDDefault defaultcode,
                              void *value,
                              unsigned valueSize);
```

Visual Basic Syntax:

```
Dim value16 As UInt16
device.GetDefault(PMDDefault.code, value16)

Dim value32 As UInt32
device.GetDefault(PMDDefault.code, value32)
```

Description:

PMDDeviceGetDefault is used to retrieve the value of a *device default*. Device defaults are various non-volatile properties of the PRP device for example the IP address, or whether to run a user program immediately after power up.

hDevice is a pointer to a handle associated with the d to retrieve the value of a *device default*. Device defaults are various non-volatile properties of the PRP device being interrogated; in C-Motion Engine user programs **hDevice** may be a null pointer, meaning the local device.

default is a numeric default code, please see the description of the **Set DefaultDevice** action in section 2.6 for a table of supported default codes and their meaning.

value is a pointer to a data area in which to store the default code, and **valueSize** is the size, in bytes, of the area. The size of a default depends on the particular data type, and is encoded in the upper byte of the **default** code – a value of zero means one byte, one means two bytes, and *n* means *n – 1* bytes. **valueSize** is required in order to prevent buffer overruns, an error code will be returned if **valueSize** is not large enough to contain the default value.

Two byte default values are generally 16-bit integers, and four byte values 32-bit integers. The **value** pointer must be properly aligned to hold these values. It is safe in all cases to require **value** to be double-word aligned, one way of accomplishing this is to use a C union type to receive the default value:

```
union defaultValue {
  PMDuint16 as_word;
  PMDuint32 as_dword;
  char as_string[32];
}
```

Related PRP Actions:

Get Device Default

Arguments:

name	**type**
hDevice	pointer to PMDDeviceHandle

C language syntax:

```
PMDresult PMDDeviceReset(PMDDeviceHandle *hDevice);
```

Visual Basic Syntax:

```
device.Reset()
```

Description:

PMDDeviceReset is used to hard reset the device, either a d to retrieve the value of a *device default*. Device defaults are various non-volatile properties of the PRP device or a Magellan attached device, associated with its argument. If it is not possible to hard reset the device then PMD_ERR_NOT_SUPPORTED will be returned. For example, Magellan attached devices controlled using CANBus, or a serial line may not be hard reset.

Related PRP Actions:

Reset Device

Reset CMotionEngine

Arguments:

name	type
hDevice	pointer to open RP device handle
defaultcode	enumerated default code
value	pointer to new default value
valueSize	size of default value

C language syntax:

```
PMDresult PMDDeviceSetDefault(PMDDeviceHandle *hDevice,
                              PMDDefault defaultcode,
                              void *value,
                              unsigned valueSize);
```

Visual Basic Syntax:

```
Dim value16 As UInt16
device.SetDefault(PMDDefault.code, value16)

Dim value32 As UInt32
device.SetDefault(PMDDefault.code, value32)
```

Description:

PMDDeviceSetDefault is used to change the value of a *device default*. Device defaults are various non-volatile properties of the PRP device, for example the IP address, or whether to run a user program immediately after power up.

hDevice is a pointer to a handle associated with the PRP device being interrogated; in C-Motion Engine user programs **hDevice** may be a null pointer, meaning the local device.

default is a numeric default code, please see the description of the **Set DefaultDevice** action in section 2.6 for a table of supported default codes and their meaning.

value is a pointer to a data area in which to store the default code, and **valueSize** is the size, in bytes, of the area. The size of a default depends on the particular data type, and is encoded in the upper byte of the **default** code – a value of zero means one byte, one means two bytes, and *n* means *n – 1* bytes. **valueSize** is required as a sanity check, an error code will be returned if **valueSize** is not large enough to contain the default value.

Two byte default values are generally 16-bit integers, and four byte values 32-bit integers. The **value** pointer must be properly aligned to hold these values. It is safe in all cases to make **value** to be double-word aligned.

Related PRP Actions:

Set Device Default

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle
major	unsigned version number
minor	unsigned version number

C language syntax:

```
PMDresult PMDDeviceGetVersion(PMDDeviceHandle *hDevice,
                              PMDuint32 *major,
                              PMDuint32 *minor);
```

Visual Basic Syntax:

```
Dim major, minor As UInteger
device.GetVersion(major, minor)
```

Description:

PMDDeviceGetVersion is used to retrieve version information for a PRP device. If **hDevice** is a handle to a Magellan attached device then **PMD_ERR_NOT_SUPPORTED** will be returned, and the version information not written. **hDevice** may be null for calls made by C-Motion Engine user programs needing the version number of the device on which they are running.

Related PRP Actions:

Get Device Version

PMDTaskGetAbortCode

C language syntax:

```
unsigned PMDTaskGetAbortCode();
```

Description:

PMDTaskGetAbortCode is used to retrieve the code left by a previous call to **PMDTaskAbort**, and may be used for communication from one instance of a C-Motion Engine user program to the next. The abort code is not non-volatile, and does not survive a reset or power cycle. After reading the abort code is cleared, and subsequent reads will return zero. Zero is also returned if **PMDTaskAbort** was not called by the previous program.

PMDTaskGetAbortCode is only available to CME user programs.

Related PRP Actions:

none

PMDDeviceGetTickCount

C language syntax:

```
PMDuint32 PMDDeviceGetTickCount();
```

Description:

PMDDeviceGetTickCount returns the number of milliseconds from the time the C-Motion Engine from which it is called has been running. The count is maintained with a granularity of 8 milliseconds, and will overflow to zero after 2^{32} milliseconds.

PMDDeviceGetTickCount is only available to CME user programs

Related PRP Actions:

none

PMDMPDeviceOpen

Arguments:

name	type
hDevice	pointer to uninitialized PMDDeviceHandle
hPeriph	pointer to PMDPeriphHandle

C language syntax:

```
PMDresult PMDMPDeviceOpen(PMDDeviceHandle *hDevice,
                          PMDPeriphHandle *hPeriph);
```

Visual Basic Syntax:

```
Dim device As New PMDDevice(peripheral,
PMDDeviceType.MotionProcessor)
```

Description:

PMDMPDeviceOpen is used to obtain a handle to a Magellan attached device, for example a non-CME ION module, or a non-CME prodigy card. A Magellan attached device communicates using the Magellan protocol, and not PRP. The ***device*** argument should point to an uninitialized PMDDeviceHandle data type, which may not be freed or written to as long as the device handle is in use.

hPeriph should point to an open peripheral connection to the Magellan attached device.

The device handle obtained using this procedure is useful for opening motion processor axis handles, using the **PMDAxisOpen** procedure.

Related PRP Actions:

Open Periph MotionProcessor

Arguments:

name	type
hMemory	pointer to open PMDMemoryHandle

C language syntax:

```
PMDresult PMDMemoryClose(PMDMemoryHandle *hMemory);
```

Visual Basic Syntax:

`memory``.Close()`

Description: **PMDMemoryClose** is used to free any resources used in maintaining a handle to a memory resource such as dual-ported RAM. After closing the memory used for the **PMDMemoryHandle** data type may be freed or re-used.

Related PRP Actions: **Close Memory**

PMDMemoryOpen

C-Motion Engine	Host-Based

Arguments:

name	type
hMemory	pointer to uninitialized PMDMemoryHandle
hDevice	pointer to PMDDeviceHandle
datasize	PMDDataType
memorytype	PMDMemoryType

C language syntax:

```
PMDresult PMDMemoryOpen32(PMDMemoryHandle *hMemory,
                          PMDDeviceHandle *hDevice,
                          PMDDataSize datasize,
                          PMDMemoryType memorytype);
```

Visual Basic Syntax:

```
Dim mem As New PMDMemory(RPDevice, PMDDataSize.Size32Bit)
```

Description:

PMDMemoryOpen is used to obtain a handle to a memory resource such as dual-ported RAM on a Prodigy/CME or non-CME Prodigy card. **hDevice** specifies the device containing the memory, and may have been opened using **PMDMPDeviceOpen** (for non-CME cards), or **PMDRPDeviceOpen** (for CME cards). In the case of C-Motion Engine user programs needing to read or write the local memory, **hDevice** should be a null pointer.

The *width* argument indicates the size of the data that are read or written to the memory device. All currently supported memory devices support only 32 bit access, so **width** must be **PMD_DataSize_32bit**. All accesses to the memory must use addresses dword-aligned, ie divisible by four, and use buffer lengths that are also divisble by four.

For all current products memorytype is one of:

PMD memoryType DPRAM
PMD memoryType DVRAM

Related PRP Actions:

Open Device Memory

Arguments:

name	type
hMemory	pointer to open PMDMemoryHandle
data	pointer to data read
offset	memory byte address
length	memory byte length

C language syntax:

```
PMDresult PMDMemoryRead(PMDMemoryHandle *hMemory,
                        void *data,
                        PMDuint32 index,
                        PMDuint32 length);
```

Visual Basic Syntax:

```
Dim offset, length As UInt32
Dim values(0 To MaxLength)
```

memory.Read(values, offset, length)

Description:

PMDMemoryRead is used to read a sequence of bytes from the memory object indicated by the *hMemory* argument. The *data* argument is a pointer to a data buffer for the values read. The *offset* argument is the memory address at which to start reading. The *length* argument is the number of bytes to read.

Each memory device has a data width, for example memory handles opened with A DATASIZE OF pmd dATAsIZE 32bIT have a data width of 4 bytes, or 32 words. If the data, offset, or *length* arguments are not aligned to the memory data width then a PMD_ERR_ALIGNMENT error code will be returned. Currently Prodigy/CME supports only dword-addressable dual-ported RAMs, and word addressable NVRAM.

Related PRP Actions:

Read Memory Dword

Arguments:

name	type
ram	pointer to open PMDMemoryHandle
data	pointer to data to write
offset	memory byte address
length	number of bytes to write

C language syntax:

```
PMDresult PMDMemoryWrite(PMDMemoryHandle *hMemory,
                         void *data,
                         PMDuint32 offset,
                         PMDuint32 length);
```

Visual Basic Syntax:

```
Dim offset, length As UInt32
Dim values(0 To MaxLength)
memory.Write(values,  offset, length)
```

Description:

PMDMemoryWrite is used to write a sequence of consecutive of bytes to the dual-ported RAM indicated by the *ram* argument. The *data* argument is a pointer to the data to write. The *offset* argument is the memory address at which to start writing. The *length* argument is the number of data units to write depending on the data size.

Each memory device has a data width. For example, memory handles opened with a datasize of PMD_DataSize_32Bit have a data width of 4 bytes, or 32 words. If the data, offset, or length arguments are not aligned to the memory data width then a **PMD_ERR_ALIGNMENT** error code will be returned. Prodigy/CME supports only dword-addressable dual-ported RAMs and word addressable NVRAM.

Related PRP Actions:

Write Memory Dword

PMDPeriphClose

Arguments:

name	type
hPeriph	pointer to open PMDPeriphHandle

C language syntax:

```
PMDresult PMDPeriphClose(PMDPeriphHandle *hPeriph);
```

Visual Basic Syntax:

```
peripheral.Close()
```

Description:

PMDPeriphClose is used to free resources associated with an open peripheral handle.

The communication channel will be closed, and no input will be accepted on it. Memory used for the peripheral handle may be freed or used for another purpose.

Related PRP Actions:

Close Peripheral

PMDPeriphOpenCAN

Arguments:

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
hDevice	pointer to open device handle
addressTx	CAN identifier for transmit
addressRx	CAN identifier for receive
eventRX	CAN identifier for event notification receive

C language syntax:

```
PMDresult PMDPeriphOpenCAN(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint32 addressTX,
                           PMDuint32 addressRX,
                           PMDuint32 eventRX);
```

Description:

PMDPeriphOpenCAN is used to open a peripheral connection to a device on a CANBus that uses two or three CAN identifiers for communication, for example a Magellan attached device or a Prodigy/CME card. **hPeriph** should point to an uninitialized **PMDPeriphHandle** data structure. **hDevice** should point to an open device handle corresponding to a PRP device, **hDevice** may be a null pointer, which means the local device, either the host or, for C-Motion Engine user programs, the local PRP device.

addressTX is a CAN identifier that will be used for sending outgoing packets. **addressRX** is a CAN identifier that will be used to listen for incoming packets. Currently only 11 bit CAN identifiers are supported.

eventRX is an optional CAN identifier used for receiving asynchronous event notification packets from a PRP device or a Magellan attached device. If no such event notification is needed then zero **eventRX** should be zero.

Related PRP Actions:

Open Device CAN

Arguments:

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
hDevice	pointer to open RP device handle

C language syntax:

```
PMDresult PMDPeriphOpenCME(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice);
```

Description:

PMDPeriphOpenCME is used to open a connection to a virtual peripheral using PRP *user packets*. User packets may contain data for user application control and monitoring in any format, but are limited in size to USER_PACKET_LENGTH (250) bytes. User packets are sent as discrete units, they do not constitute a stream.

User packets are transported in PRP packets, that is, they are "tunneled" through PRP, and are a very simple way to establish communication between host programs and C-Motion engine user programs because they do not require opening a separate communication channel, nor implementing a low-level protocol over it.

PMDPeriphOpenCME is used to open both sides of the user packet channel: On the host side an opened device handle associated with a PRP device must be passed using the ***hDevice*** argument. On the C-Motion engine side a user program should pass a null pointer as ***hDevice***.

The peripheral handle opened by **PMDPeriphOpenCME** may be used in the same way as other peripheral handles, using **PMDPeriphSend**, **PMDPeriphReceive**, and **PMDPeriphClose**.

When considering the timeout parameter for peripheral send and receive commands for user packets, it is useful to know that the C-Motion Engine can buffer one user packet on the incoming side, and one on the outgoing side. The timeout period is not determined by when something actually reads a user packet, but rather by when it is copied into the appropriate buffer. There are four cases to consider:

1. A host sending user packets to a CME can always send one packet without a timeout, but the *second* packet will time out if a CME user program has not read the first packet in the specified time.
2. A host receiving user packets from a CME will time out if a CME user program has not written a packet to the outgoing buffer by the specified time.
3. A CME sending user packets to a host can always send one packet without a timeout, but the *second* packet will time out if a host program has not read the first packet in the specified time.
4. A CME receiving user packets will time out if a host program has not written a user packet to the incoming buffer in the specified time.

While it is possible for multiple host processes or multiple hosts to read and write user packets to the same PRP device, but it is not a good idea. There is no way to determine which host sent a given packet, nor any way to "unread" or "peek" at an incoming user packet.

Related PRP Actions:

Open Device CMotionEngine

Send CMotionEngine

Receive CMotionEngine

PMDPeriphOpenCOM

C-Motion Engine | Host-Based

Arguments:

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
hDevice	pointer to RP device handle
port	enumerated serial port
baud	enumerated baud rate
parity	enumerated parity
stopbits	enumerated number of stop bits

C language syntax:

```
PMDresult PMDPeriphOpenCOM(PMDPeriphHandle *periph,
                           PMDDeviceHandle *device,
                           PMDSerialPort port,
                           PMDSerialBaud baud,
                           PMDSerialParity parity,
                           PMDSerialStopBits stopbits);
```

Visual Basic Syntax:

```
Dim periph As New PMDPeripheralCOM(portnum, PMDSerialBaud.baud, _
                      PMDSerialParity.parity, PMDSerialStopBits.bits)
```

Description:

PMDPeriphOpenCOM is used to open a peripheral handle representing an open serial line. **hPeriph** should point to an uninitialized **PMDPeriphHandle** data structure. **hDevice** is a device handle which should be associated with a PRP device, **hDevice** may be a null pointer, in which case it means the local device, either the host or, for a C-Motion Engine user program, the local PRP device.

port is the serial port to use, one of **PMDSerialPort1** or **PMDSerialPort2**.

baud is the serial port speed to set, one of **PMDSerialBaud1200**, **PMDSerialBaud2400**, **PMDSerialBaud9600**, **PMDSerialBaud19200**, **PMDSerialBaud57600**, **PMDSerialBaud115200**, **PMDSerialBaud230400**, or **PMDSerialBaud460800**.

parity is the parity to use, one of **PMDSerialParityNone**, **PMDSerialParityOdd**, or **PMDSerialParityEven**.

stopbits is the number of stopbits to use, either **PMDSerialStopBits1** or **PMDSerialStopBits2**.

Eight data bits are always used.

In order to open a PMD serial protocol multi-drop peripheral, **PMDPeriphOpenMultiDrop** should be applied to the peripheral handle opened by **PMDPeriphOpenCOM**.

Related PRP Actions:

Open Device COM

Arguments:

name	type
hPeriph	pointer to uninitialized peripheral handle
hDevice	pointer to open RP device handle
address	ISA base address
eventIRQ	ISA interrupt line
width	enumerated data size

C language syntax:

```
PMDresult PMDPeriphOpenISA(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint16 address,
                           PMDuint8 eventIRQ,
                           PMDDataSize width);
```

Description:

PMDPeriphOpenISA is used to open a peripheral representing a device on the PC-104

ISA bus at a specified base **address**. **hPeriph** should point to an uninitialized PMDPeriphHandle, and **hDevice** should be a pointer to an open RP device handle, that is, a PRP device. If called from a C-Motion Engine user program then **hDevice** may be a null pointer, meaning the local device.

The **PMDPeriphReadBytes** and **PMDPeriphWriteBytes** procedures may be used to read or write to the ISA bus at specified offsets from the base **address**.

In case the peripheral is connected to a non-CME Prodigy card then **eventIRQ** may be used to specify the interrupt used for asynchronous event notification.

The ***width*** argument specifies the size of the data that are read or written to the peripheral. Non-CME Prodigy-ISA cards require 16-bit data access, so **width** should be **PMD_DataSize_16bits** when opening such a device. ISA devices requiring 8-bit access are also supported, and use the value **PMD_DataSize_8bits** for **width**.

All reads or writes to a 16-bit ISA peripheral must be properly aligned, that is, all address values data lengths must be even.

Related PRP Actions:

Open Device ISA

PMDPeriphOpenMultiDrop

C-Motion Engine	Host-Based

Arguments:

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
hParent	pointer to open handle to serial port peripheral
address	5 bit PMD multi-drop address

C language syntax:

```
PMDresult PMDPeriphOpenMultiDrop(PMDPeriphHandle *periph,
                                 PMDPeriphHandle *parent,
                                 unsigned address);
```

Visual Basic Syntax:

```
Dim parent As PMDPeripheralCOM
Dim address As UInt32
Dim periph As New PMDPeripheralMultiDrop(parent, address)
```

Description: PMDPeriphOpenMultiDrop is used to open a peripheral representing a connection on a serial line to a device using the PMD multi-drop serial protocol, either a Magellan attached device or a PRP device. hParent must be a pointer to a previously opened peripheral representing the serial line, and address is the multi-drop address.

Related PRP Actions: **Open Peripheral MultiDrop**

Arguments:

name	type
hPeriph	pointer to uninitialized peripheral handle
hDevice	pointer to a valid device handle
address	16 bit address indicating peripheral channel to open
EventIRQ	Device-specific interrupt channel
datasize	Data width of the peripheral in bytes

C language syntax:

```
PMDresult PMDPeriphOpenPAR(
                                PMDPeriphHandle* hPeriph,
                                PMDDeviceHandle *hDevice,
                                WORD address,
                                BYTE EventIRQ,
                                PMDDataSize datasize);
```

Description:

PMDPeriphOpenPAR is used to open a peripheral handle representing a parallel channel on the indicated device. The nature of the parallel channel is specific to the device being addressed. Currently ION/CME supports parallel channels used for digital input and output and for analog input.

The address argument indicates the specific parallel channel to be opened, and is device-specific. The datasize argument indicates the data width of the peripheral to be opened, that is, the number of 8 bit bytes read or written with each operation. Only one data width is normally supported for each type of parallel channel. The **EventIRQ** argument indicates the interrupt used for parallel communication, and is device-specific.

Currently only the ION/CME digital drive supports parallel peripherals, which are used for digital input/output and for analog input. Consult the *ION/CME Digital Drive User's Manual* for details.

Related PRP Actions:

Open Device PAR

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
cardNo	integer

C language syntax:

```
PMDresult PMDPeriphOpenPCI(PMDPeriphHandle *hPeriph,
                           int cardNo
```

Visual Basic Syntax:

```
Dim boardnum As UInt32
Dim periph As New PMDPeripheralPCI(boardnum)
```

Description: **PMDPeriphOpenPCI** is used on a host PC to open a peripheral connection to a Prodigy/CME-PCI card installed in the host computer. Because Prodigy/CME-PCI does not support bus mastering there is no way of opening an outgoing PCI bus peripheral on the Prodigy/CME. cardNo is a small integer denoting the particular Prodigy/CME card to connect to. If only one Prodigy/CME card is present, then cardNo is always zero. Mutiple cards are numbered sequentially in an order that must be determined by experiment.

Related PRP Actions: none, this procedure is supported only on a PC host.

Arguments:

name	type
hPeriph	pointer to uninitialized PMDPeriphHandle
hDevice	pointer to open PMDDeviceHandle
IPAddress	32 bit IP address
port	16 bit TCP/IP port

C language syntax:

```
PMDresult PMDPeriphOpenTCP(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint32 IPAddress,
                           PMDuint16 port);
```

Visual Basic Syntax:

```
Dim address As System.Net.IPAddress
Dim portnum, timeout As UInt32
Dim periph As New PMDPeripheralTCP(address, portnum, timeout)
```

Description:

PMDPeriphOpenTCP is used to open a TCP/IP peripheral on the PRP device indicated by **hDevice**. If **hDevice** is a null pointer then the local device, either the host or the PRP device on which a CME user program is running.

If **IPAddress** is nonzero then it is the IP address of a remote Ethernet device to which a connection should be opened. If **IPAddress** is nonzero then the device will listen on the indicated TCP **port** for incoming connections from any device, handle one connection at a time, and resume listening after a remote device closes the connection. In either case, a connection may be closed using **PMDPeriphClose**.

IPAddress must be numeric, PRP devices do not support any kind of name service. An IP address in the familiar dotted quad notation **A.B.C.D** is equivalent to the 32 bit number **(A<<24) + (B<<16) + (C<<8) + D**, this conversion may be done using the macro PMD_IP4_ADDR, for example the numeric value of the IP address 192.168.13.42 could be obtained by writing **PMD_IP4_ADDR(192, 168, 13, 42)**.

port is the TCP port number to use for sending or receiving. TCP ports are divided into three ranges:

1. The *well-known* ports from 0 to 1023 are used for standard services, which are not likely to be hosted by user C-Motion Engine applications.
2. The *registered ports* from 1024 to 49151 are used *ad hoc*, and are most likely to be used for user motion control applications,
3. The dynamic ports from 49152 to 65535 are used for temporary applications, and may be useful for user applications that dynamically assign UDP ports.

Related PRP Actions:

Open Device TCP

PMDPeriphOpenUDP

Arguments:

name	**type**
hPeriph	pointer to uninitialized PMDPeriphHandle
hDevice	pointer to open PMDDeviceHandle
IPAddress	32 bit IP address
port	16 bit UDP port

C language syntax:

```
PMDresult PMDPeriphOpenUDP(PMDPeriphHandle *hPeriph,
                           PMDDeviceHandle *hDevice,
                           PMDuint32 IPAddress,
                           PMDuint16 port);
```

Description:

PMDPeriphOpenUDP is used to open a UDP/IP peripheral on the PRP device indicated by **hDevice**. If **hDevice** is a null pointer then the local device, either the host or the PRP device on which a CME user program is running.

If **IPAddress** is nonzero then it is the IP address of a remote Ethernet device to which packets will be sent; the peripheral will be write-only. If **IPAddress** is zero then a UDP port will be opened for listening; the peripheral will be read-only. **IPAddress** must be numeric, PRP devices do not support any kind of name service. An IP address in the familiar dotted quad notation **A.B.C.D** is equivalent to the 32 bit number **(A<<24) + (B<<16) + (C<<8) + D**, this conversion may be done using the macro PMD_IP4_ADDR, for example the numeric value of the IP address 192.168.13.42 could be obtained by writing **PMD_IP4_ADDR(192, 168, 13, 42)**.

port is the UDP port number to use for sending or receiving. UDP ports are divided into three ranges:

1. The *well-known* ports from 0 to 1023 are used for standard services, which are not likely to be hosted by user C-Motion Engine applications.
2. The *registered ports* from 1024 to 49151 are used *ad hoc*, and are most likely to be used for user motion control applications,
3. The dynamic ports from 49152 to 65535 are used for temporary applications, and may be useful for user applications that dynamically assign UDP ports.

Related PRP Actions:

Open Device UDP

Arguments:

name	**type**
hPeriph	pointer to open PMDPeriphHandle
data	buffer for incoming data
offset	byte offset from base address
length	number of data units to read

C language syntax:

```
PMDresult PMDPeriphRead (PMDPeriphHandle *hPeriph,
                         void *data,
                         PMDuint32 offset,
                         PMDuint32 length);
```

Visual Basic Syntax:

```
Dim data16(0 To MaxLength) As UInt16
Dim data8(0 To MaxLength) As Byte
Dim offset, length As UInt32
periph.read(data16, offset, length)
periph.read(data8, offset, length)
```

Description:

PMDPeriphRead is used to read a stream of bytes from a peripheral with a specified base address, specifically PC-104 ISA bus and PCI bus peripherals. **hPeriph** should point to an open handle to such a peripheral, for peripherals without an address concept an error code of **PMD_ERR_NOT_SUPPORTED** will be returned.

data is a pointer to a buffer for incoming data, **offset** is an increment to add to the base address to give the address to read from, and **length** is the number of bytes to read.

Related PRP Actions:

Read Periph Byte

Arguments:

name	type
hPeriph	pointer to open PMDPeriphHandle
data	pointer to incoming data buffer
nReceived	pointer to actual bytes received
nExpected	maximum bytes to receive
timeout	milliseconds, less than 0xffff

C language syntax:

```
PMDresult PMDPeriphReceive(PMDPeriphHandle *periph,
                           void *buffer,
                           PMDuint32 *nReceived,
                           PMDuint32 nExpected,
                           PMDuint32 timeout);
```

Visual Basic Syntax:

```
Dim data8(0 To MaxLength) As Byte
Dim nReceived, nExpected, timeout As UInt32
periph.receive(data8, nReceived, nExpected, timeout)
```

Description:

PMDPeriphReceive is used to read bytes from a peripheral. **hPeriph** should be a pointer to an open peripheral handle, **data** a pointer to a memory buffer for incoming data, and **nExpected** the maximum number of bytes to accept, typically the size of the **data** buffer.

For peripherals that receive data in packets, such as CANBus, TCP/IP, and UDP/IP, **PMDPeriphReceive** will return after receiving one packet, writing to the **data** buffer, and writing the actual number of bytes received to ***nReceived**. Note that the number of bytes received may be greater than **nExpected**, but at most **nExpected** bytes will be written in the buffer.

For peripherals that do not receive data in packets, such as serial ports, **PMDPeriphReceive** will return after receiving exactly **nExpected** bytes.

PMDPeriphReceive will return **PMD_RP_Timeout** if **timeout** milliseconds elapsed waiting for data. Some ports may timeout before receiving nExpected bytes. The nReceived parameter will contain the number of bytes received before the timeout. A **timeout** value of **PMD_WAITFOREVER** (0xffff) disables the time out.

If the peripheral connection has been closed by some external action, for example a TCP connection that has been closed by a peer, then **PMD_ERR_NotConnected** will be returned. After such an error the peripheral handle must be closed using **PMDPeriphClose**. In the case of a TCP connection, after closing the unconnected peripheral a new peripheral with the same TCP port may be opened using **PMDPeriphOpenTCP**.

The following example shows how to implement a TCP server that handles a single connection at a time, and reads data until the connection is closed by the peer.

```
PMDresult status;
PMDPeriphHandle hPeriphTCP;
PMDuint32 nReceived;
unsigned char buffer[PACKETSIZE];
int open;

while (!0) {
   status = PMDPeriphOpenTCP(&hPeriphTCP, NULL, 0, TCPPORT, timeout);
   open = 1;
```

```
        while (open) {
            status = PMDPeriphReceive(&hPeriphTCP, buffer, &nReceived, sizeof(buffer), 
timeout);
            // As a simple example we just read data. For a more complicated protocol each send and
            // receive operation should include a check of the return value as shown.
            switch (status) {
            default:
                Handle the error;
            case PMD_ERR_NotConnected:
              // The peripheral handle must be closed. It will be re-opened in the outer loop.
                PMDPeriphClose(&hPeriphTCP);
                open = 0;
                break;
            case PMD_ERR_OK:
                Do something useful with the data;
                break;
            }
        }
    }
```

Related PRP Actions: **Receive Peripheral**

PMDPeriphSend

C-Motion Engine | Host-Based

Arguments:

name	type
hPeriph	pointer to open PMDPeriphHandle
data	pointer to data to send
nCount	number of bytes to send
timeout	milliseconds to wait, less than 0xffff

C language syntax:

```
PMDresult PMDPeriphSend(PMDPeriphHandle *hPeriph,
                        void *data,
                        PMDuint32 nCount,
                        PMDuint32 timeout);
```

Visual Basic Syntax:

```
Dim data8(0 To MaxLength) As Byte
Dim nCount, timeout As UInt32
periph.receive(data8, nCount, timeout)
```

Description:

PMDPeriphSend is used to send bytes to a peripheral, indicated by the ***hPeriph*** argument.

nCount bytes are sent from the **buffer** data. If the data may not be sent in **timeout** milliseconds then **PMDPeriphSend** will stop trying and return **PMD_ERR_Timeout**. A **timeout** value of **PMD_WAITFOREVER** (0xffff) means never stop trying.

If the peripheral connection has been closed by some external action, for example a TCP connection that has been closed by a peer, then **PMD_ERR_NotConnected** will be returned. After such an error the peripheral handle must be closed using **PMDPeriphClose**. In the case of a TCP connection, after closing the unconnected peripheral a new peripheral with the same TCP port may be opened using **PMDPeriphOpenTCP**. See **PMDPeriphReceive** (p. 107) for example code.

Related PRP Actions:

Send Peripheral

Arguments:

name	type
hPeriph	pointer to an open peripheral handle
data	pointer to data to write
offset	offset from base address
length	number of data units to write

C language syntax:

```
PMDresult PMDPeriphWrite(PMDPeriphHandle *hPeriph,
                         void *data,
                         PMDuint32 offset,
                         PMDuint32 length);
```

Visual Basic Syntax:

```
Dim data16(0 To MaxLength) As UInt16
Dim data8(0 To MaxLength) As Byte
Dim offset, length As UInt32
periph.read(data16, offset, length)
periph.read(data8, offset, length)
```

Description:

PMDPeriphWrite is used to write a stream of bytes to a peripheral with a specified base address, specifically PC-104 ISA bus and PCI bus peripherals. **hPeriph** should point to an open handle to such a peripheral, for peripherals without an address concept an error code of **PMD_ERR_NOT_SUPPORTED** will be returned.

data is a pointer to a buffer containing the data to write, **offset** is an increment to add to the base address to give the address for writing, and **length** is the number of bytes to write.

Related PRP Actions:

Write Periph Byte

Arguments:

name	type
hIO	pointer to uninitialized PMDIOHandle
hDevise	pointer to PMDDeviceHandle
iotype	iotype identifier

C language syntax:

```
PMDresult PMDIOOpen(PMDIOHandle *hIO,
                      PMDDeviceHandle *hDevice,
                      PMDIOType iotype);
```

VisuaDescription:

PMDIOOpen is used to obtain a handle to an IO resource on an ION/CME module. **hDevice** specifies the device containing the IO resource. In the case of C-Motion Engine user programs needing to read or write to the local IO, **hDevice** should be null.

For all current products iotype can be of:

PMDIOType DIO = 0
PMDIOType AI = 1

Arguments:

name	type
hIO	pointer to an open PMDIOHandle
data	pointer to data

C language syntax:

```
PMDresult PMDIORead(PMDIO *IO,
                     PMDuint16 *data);
```

VisuaDescription:

PMDIORead is used is used to read the value at an IO port indicated by the hIO argument. The data argument is a pointer to a 16-bit value to receive value read.

Related PRP Actions:

Read IO Word

PMDIOWrite

Arguments:

name	type
hIO	pointer to an open PMDIOHandle
data	16-bit data to be written
mask	digital IO mask

C language syntax:

```
PMDresult PMDIOWrite(PMDIO *hIO,
                     PMDuint16 data,
                     PMDuint16 mask);
```

VisuaDescription:

PMDIOWrite is used to write a value to a general purpose IO port (digital or analog). If the IO port type being addressed is a digital IO port, the mask field is used to specify which bits will be affected by the write.

Related PRP Actions:

Write IO Word

PMDprintf

Arguments:

name	**type**
fmt	string
…	arguments to format

C language syntax:

```
int PMDprintf(const char *fmt, …);
```

Description:

PMDprintf is the primary procedure used for console output, a feature used for progress reporting during development and debugging. The console may be attached to any of the available communication devices at startup using the default settings **Default_DebugIntfType**, **Default_DebugIntfAddr**, and **Default_DebugIntfPort**. The console may be changed at run time to a specified peripheral by using the PRP action **Set Console.** Pro-Motion can also be used conveniently to set the current or default console.

The arguments to **PMDprintf** are the same as to the C standard library **printf**, and the return value is the number of characters printed. Because there is only one console and no file system there is no equivalent to **fprintf**. In order to send formatted data through a peripheral **sprintf** should be used to format to a user-supplied buffer, and the buffer sent.

PMDprintf does not correctly format floating point arguments. In order to print floating point numbers it is necessary to format them using **sprintf**, and then to print the formatted string using **PMDprintf** or **PMDputs**.

Related PRP Actions:

Set Console

Set Device Default Default_DebugIntfType

Set Device Default Default_DebugIntfAddr

Set Device Default Default_DebugIntfPort

Arguments:

name	type
ch	8 bit integer

C language syntax:

```
void PMD_putch(int ch);
```

Description:

PMDputch is used to print a single character to the console. See also **PMDprintf** (p. 111) for more description of the console.

Related PRP Actions:

Set Console

Set Device Default Default_DebugIntfType

Set Device Default Default_DebugIntfAddr

Set Device Default Default_DebugIntfPort

Arguments:

name	**type**
str	string

C language syntax:

```
void PMDputs(const char *str);
```

Description:

PMDputs is used to print a constant string to the console. See also PMDprintf (p. 111) for more description of the console.

Related PRP Actions:

Set Console

Set Device Default Default_DebugIntfType

Set Device Default Default_DebugIntfAddr

Set Device Default Default_DebugIntfPort

PMDRPDeviceOpen

Arguments:

name	type
hDevice	pointer to uninitialized PMDDeviceHandle
hPeriph	pointer to open PMDPeriphHandle

C language syntax:

```
PMDresult PMDRPDeviceOpen(PMDDeviceHandle *hDevice,
                          PMDPeriphHandle *hPeriph);
```

Visual Basic Syntax:

```
Dim dev As New PMDDevice(periph, PMDDeviceType.ResourceProtocol)
```

Description:

PMDRPDeviceOpen is used to open a handle to a device that communicates using PRP, that is, a Prodigy/CME card or PRP ION module. **hPeriph** should be a handle to an open peripheral that is physically connected to a PRP device.

The device handle opened by this procedure may be used for opening motion processor axes, (see **PMDAxisOpen** (p. 81)), or dual-ported RAM devices (see **PMDMemoryOpen32** (p. 94)), peripherals on the device (see **PMDPeriphOpenCOM** (p. 100), **PMDPeriphOpenTCP** (p. 104), **PMDPeriphOpenUDP** (p. 105), **PMDPeriphOpenISA** (p. 101), and **PMDPeriphOpenCAN** (p. 98)) .

The device handle is also used to access the C-Motion Engine on the device, for example using **PMDCMETaskStart** or **PMDCMETaskStop**.

Related PRP Actions:

Open Peripheral Device

PMDTaskAbort

Arguments:

name	type
UserAbortCode	8 bit integer

C language syntax:

```
void PMDTaskAbort(int UserAbortCode);
```

Description:

PMDTaskAbort is used to halt user code execution in case of a fatal error, it does not return. The argument is a nonzero code that can be used to communicate the cause of failure to the next invocation of the user program, and should be checked using **PMDTaskGetAbortCode** at the beginning of the user program.

PMDTaskAbort does not perform any cleanup actions, nor does it perform a reset. Any cleanup required to put the device in a safe state must be done by the user program before calling **PMDTaskAbort**.

Related PRP Actions:

none. This procedure may be called only from a C-Motion Engine user program.

PMDTaskWait

Arguments:

name	type
msec	milliseconds

C language syntax:

```
void PMDTaskWait(PMDuint32 msec);
```

Description:

The **PMDTaskWait** procedure is used to delay execution of a C-Motion Engine user program for a specified number of milliseconds. The delay is relative to the time the procedure is called, and has a granularity of 8 milliseconds.

For a way to arrange a periodic task, see PMDTask**WaitUntil** (p. 118).

Related PRP Actions:

none

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle
hEvent	pointer to event struct
timeout	milliseconds, up to 0xfffe

C language syntax:

```
PMDresult PMDWaitForEvent(PMDDeviceHandle *hDevice,
                          PMDEvent *hEvent,
                          PMDuint32 timeout);
```

Visual Basic Syntax:

```
Dim EventStruct As PMDEvent
Dim timeout As UInt32
device.WaitForEvent(EventStruct, timeout)
Dim axis As PMDAxis
Dim EventMask As UInt16
axis = EventStruct.axis
EventMask = EventStruct.EventMask
```

Description:

PMDWaitForEvent is used to check for any reported asynchronous events raised by the device indicated by **hDevice**. The device may be a Magellan attached device, a PRP device, or, for C-Motion Engine user programs, the null pointer to indicate the local device.

If an asynchronous event notification is received for any of the Magellan axes of the motion processor attached to the device then **PMD_ERR_RP_EVENT** will be returned and the axis and event status register written to members of the **hEvent** struct. This struct has at least these members:

```
PMDAxis axis;
PMDuint16 eventStatus;
```

which indicate the axis and events responsible for the notification. If no event notifications have been received within **timeout** milliseconds, then zero (success) is returned, and **hEvent** is not written. A **timeout** value of **PMD_WAITFOREVER** (0xffff) disables the time out.

Asynchronous event notification is an optional Magellan feature described in the *Magellan Motion Processor User's Guide*. The conditions causing an event notification are programmable, using commands described in the *Magellan Motion Processor Programmer's Reference.* Not all peripheral types support event notification, in particular serial communication does not. All peripherals in the chain used to communicate with a given motion processor must have been opened with the appropriate event channel data in order for event notification to work.

Related PRP Actions:

See the description of the Event Notification Packet, section 2.4.3.

PMDTaskWaitUntil

Arguments:

name	type
pPreviousTime	pointer to time in milliseconds
incrms	increment in milliseconds

C language syntax:

```
void PMDTaskWaitUntil(PMDuint32 *pPreviousTime, PMDuint32 incrms);
```

Description:

The **PMDTaskWaitUntil** procedure is used to wait until a particular specified time and may be used to arrange a periodic task loop. The argument ***pPreviousTime*** should point to a timer count previously returned by **PMDDeviceGetTickCount** or modified by **PMDTaskWaitUntil**. **PMDTaskWaitUntil** will return after the timer tick computed by adding **incrms** to the tick value in ***pPreviousTime**. The value in ***pPreviousTime** will be updated to the current time.

If the time computed by adding **incrms** to ***pPreviousTime** is in the past then

PMDTaskWaitUntil will return immediately and will not update ***pPreviousTime**. If this case is likely, it must be checked explicitly using **PMDDeviceGetTickCount**.

For example:

```
PMDuint32 lastTime, thisTime;
PMDuint32 incrTime = 32;

lastTime = PMDDeviceGetTickCount();
while (!0) {

   Do some useful job

   thisTime = PMDDeviceGetTickCount();
   if ((lastTime + incrTime < thisTime) &&
       (lastTime + incrTime > lastTime)) {
          Report a time budget overrun
       lastTime = thisTime;
   }
   PMDTaskWaitUntil(*lastTime, incrTime);   // wait for up to 32 milliseconds
```

Related PRP Actions:

none

5. C-Motion Engine

The C-Motion Engine is a special purpose computer included in the Prodigy/CME card and the ION/CME digital drive, and connected by a high speed internal bus to the on-card Magellan Motion Processor, dual-ported RAM, and various communication devices. The firmware libraries required for motion control and a framework for application support are already included in the CME device, only the logic specific to a particular application need be programmed into the C-Motion Engine, making development a much quicker task than it would be for a “ground-up” embedded application.

Most of the instruction cycles in the microprocessor hosting the C-Motion Engine are normally available for running the user program, but processing of messages sent and received on communication peripherals is done by the same processor. Heavy message traffic, particularly heavy Ethernet traffic, may therefore reduce the time available for running the user program.

5.1 C-Motion Libraries

The C-Motion Engine is programmed in ANSI C89, and supports the applicable parts of the standard C library and the math library. Standard I/O is limited to the pre-opened “stderr” and “stdout” file handles, which are connected to the console, because there are no available devices supporting files. Floating point is available but is implemented in software. There is a significant performance penalty compared to integer arithmetic, so fixed point computation should be considered in applications for which speed is important.

Dynamic memory allocation is supported using “malloc” and “free.” Because the dynamic heap is of limited size and is unavoidably subject to fragmentation it is suggested that dynamic allocation be used sparingly, preferably only during initialization. The heap in current CME devices is approximately 7 kilobytes.

5.2 Procedures Specific to the C-Motion Engine

Most of the Procedures used for controlling aCME device use identical calling sequences whether executed on the host or the C-Motion Engine, making it easy to move parts of an application from the host to the C-Motion engine, or vice-versa. See Section 4.5, PMD Library Procedures, for the internals of the data structures used, and other details of implementation are of course different. Procedures specific to the C-Motion engine.

C language procedures dealing with an entire PRP device, for example to open a handle to a card resource, use a device handle, a pointer to the **PMDDevice** type. For host-based programs this type must be initialized by a call to **PMDRPDeviceOpen** to establish a connection to a PRP device over an open peripheral connection. For C-Motion engine user programs accessing the local device, a null pointer should be passed as the device handle. An example is shown below:

```
/* Host-based program */
PMDDeviceHandle dev;        /* Statically allocate space for a PRP device handle. */
PMDAxisHandle axis1;   /* Allocate space for a handle to a Magellan axis. */
PMDPeriphHandle periph;      /* Allocate space for a handle to a peripheral connection. */
```

```
int main(int argc, char **argv)
{
    PMDresult result;
    PMDDeviceHandle *hdev = &dev;

   /* Open a TCP/IP peripheral, the second argument is the device handle, which is
       zero because the peripheral connection is to the host. */
result = PMDPeriphOpenTCP(&periph, 0, IP_ADDRESS, TCP_PORT, 0);
   if (result) {  Handle the error  }

  /*   Establish a connection to a PRP device over the opened peripheral. */
result = PMDRPDeviceOpen(hdev, &periph);
  if (result) {  Handle the error }

result = PMDAxisOpen(&axis1, hdev, PMDAxis1);
  if (result) { Handle the error }

/* Now do something useful with the Magellan axis just opened. */
     …
}

/* -----------------------------------------------------------------------------------------------*/
/* C-Motion Engine program */
PMDDeviceHandle dev;         /* Statically allocate space for a Prodigy/CME device handle. */
PMDAxisHandle axis1;   /* Allocate space for a handle to a Magellan axis. */

USER_CODE_TASK(myTask)
{
    PMDresult result;
    PMDDeviceHandle *hdev = 0;         /* Use a null pointer for the local device handle. */

  /* Open a handle to a Magellan axis on the local device. */
result = PMDAxisOpen(&axis1, hdev, PMDAxis1);
    if (result) {  Handle the error }

/* Now do something useful with the Magellan axis just opened. */
    …
}
```

5.3 C-Motion Engine Programming

In many ways the C-Motion engine environment is more restrictive than a PC host environment: code size, data size, and stack size are all more limited (see the User's Guide for your product). The processor running the C-Motion Engine is slower than a typical PC processor, but because of the lack of competing processes it can be much more predictable and quicker to respond.

The programming model of the C-Motion Engine has been deliberately kept simple. A C-Motion Engine user program has a single thread of control, and must not exit, except for fatal error processing using the **PMDTaskAbort** procedure. There is no support for user interrupt handling, multithreading, or other asynchronous processing, so user programs are generally structured as event loops that poll for every type of event that they must handle.

Typically the heart of a user program is a state machine, which, depending on the current state, initiates particular actions and checks for specific input in order to determine the next state. The following provides illustrates a typical example, a function that handles one state transition, and returns the next state.

```
int nextState(PMDPeriphHandle *hCmdPeriph, enum myStates state)
{
    PMDResult result;
    PMDuint32 n;

    if (General error condition)
        return STATE_ERROR;

    switch (state) {
    default:
        return STATE_ERROR; /* Should not happen. */

    case STATE_WAIT_FOR_COMMAND:
        result = PMDPeriphReceive(hCmdPeriph, buf, &n, sizeof(buf), 0);
        if (0 == result) {
            /* We got a message, parse it and return the next state.  What parseCommand does depends
                on the command language we're using. */
            return parseCommand(buf, n);
        }
        else if (PMD_ERR_RP_Timeout == result) {
            /* No message, keep waiting. */
            return state;
        }
        else {
            /* We got an error, report it and move on. */
            return state;
        }

    case STATE_EXTEND_ARM_0:
        Start step 1
        return STATE_EXTEND_ARM_1:

    case STATE_EXTEND_ARM_1:
        if (Step 1 done) {
            Start step 2;
            return STATE_EXTEND_ARM_2;
        }
        else
            return state;

    case STATE_EXTEND_ARM_2:
        if (Step 2 done)
            return STATE_WAIT_FOR_COMMAND;
        else
            return state;

    case ...


    }
}
```

In practice **nextState** would be called in a non-terminating loop. Exactly how this is done depends partly on the time constraints of the user task. One approach is just to run as fast as possible:

```
USER_CODE_TASK(myTask)
{
    enum myState state = STATE_WAIT_FOR_COMMAND;

    Initialization code ...;

    while (1) {
        state = nextState(state);
    }
}
```

Another approach is to schedule the state transitions at a regular interval, which has the advantage of allowing one to find out during testing and debugging that the expected time budget has been exceeded:

```
USER_CODE_TASK(myTask)
{
    enum myState state = STATE_WAIT_FOR_COMMAND;
    PMDresult result;
    PMDuint32 lastTime, thisTime;
    PMDuint32 incrTime = 50;    /* Allow 50 milliseconds for each state transition. */

    Initialization code ...;

    lastTime = PMDGetTickCount();
    while (1) {
        state = nextState(state);
        thisTime = PMDDeviceGetTickCount();
        if ((lastTime + incrTime < thisTime) &&
            (lastTime + incrTime > lastTime)) {

            Report time budget exceeded;

            lastTime = thisTime;
        }
      PMDTaskWaitUntil(&lastTime, incrTime);
    }
}
```

5.4 The Console

The C-Motion Engine console is a primary tool for reporting progress and error messages during development. By default the console is attached to serial port 2, but it may be redirected to a UDP/IP port or disabled by redirecting it to the null peripheral. The most convenient way of doing this during development is by using Pro-Motion, as described in the *Pro-Motion User's Guide*. **PMDprintf**, **PMDputs**, and **PMDputch** may be used to send formatted messages to the console. See Section 4.5, PMD Library Procedures, for procedure documentation.

5.5 User Packets

User packets are a convenient method of encapsulating arbitrary data in the control stream used by PMD procedures for communication from the host to a CME device. User packets are likely to be most useful for prototypes applications, but if performance requirements are modest they may be useful for deployed applications.

Each end of the user packet stream is represented by a peripheral handle, obtained using the **PMDPeriphOpenCME** procedure (see Section 4.5, PMD Library Procedures,). This procedure is available both on the host and the C-Motion Engine, and in each case returns one end of the stream. A host program might send application commands as user packets in any desired format, and the C-Motion Engine user program could reply with user packets representing status information.

It is important to remember that user packets are delivered as discrete packets, with a maximum size of 250 bytes, and that only one incoming and one outgoing user packet may be buffered in the C-Motion Engine at any time – if several user packets are sent without waiting for receipt then all but the first may be overwritten.

5.6 Exceptions

In case of an unrecoverable error a C-Motion user program may call the **PMDTaskAbort** procedure, which will result in the user program exiting. The user program must leave the CME device and any attached hardware in a safe state before calling this procedure. The user program may be restarted using Pro-Motion, and may, during startup, check for an error code left by **PMDTaskAbort** using the **PMDTaskGetAbortCode** procedure. The error code is not non-volatile, it does not survive a reset or power cycle.

An error in a C-Motion user program may cause a processor exception, for example by reading or writing from an invalid address, or attempting to execute an invalid instruction (typically because of stack corruption or an invalid function pointer). After an exception the microprocessor hosting the C-Motion Engine will be reset by the system watchdog, and execution of the user program will be suppressed regardless of the value of the **DefaultAutoStartMode** device default (see **Set Device ValueDefault** in Section 3.3, PRP Actions and Sub-Actions, for more information).

When a CME device restarts after a system watchdog reset, information on any previous exception will be printed to the console, including register values and stack contents. With the aid of the program memory map, this may be useful in debugging the exception.

This page intentionally left blank.

6. C-Motion

C-Motion is the PMD library for control of Magellan Motion Processors, and is documented in the *Magellan Motion Processor Programmer's Command Reference.* A version of C-Motion is provided for controlling the motion processor included in a CME device from a host computer. There is also a C-Motion library with the same callable procedures that runs in the C-Motion Engine.

6.1 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:

- Version 4.x, for host programs that communicate with PRP devices, and for C-Motion Engine programs.
- Version 3.x, for all other PMD Magellan products, including non-PRP ION modules, non-CME Prodigy cards, and Magellan Motion Processors.

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, and they do not share the same mechanisms for opening a connection to a Motion Processor, discussed for Version 4.x in section 5.2 Axis Handles.

C-Motion 3.x requires the communication interface (PCI, ISA, serial, or CAN) to be specified at compile time. This allows a smaller program, and, in the case of a port to a microprocessor host, means that code for interfaces that are not used need not be ported. C-Motion 3.x uses only the Magellan protocols, and does not support PRP.

C-Motion 4.x allows the communications interface (PCI, TCP, serial, or CAN) to be specified at run-time, and supports multiple connections using different interfaces at the same time. A larger and more complex library is therefore required. C-Motion 4.x supports both the Magellan protocols, which are used to communicate with Magellan attached Motion Processors, and also PRP, which is used to communicate with Prodigy/CME cards and ION/CME and ION/D digital drives. A port of C-Motion 4.x to a microprocessor host could certainly omit some interfaces, but source code changes in various parts of the library would be required.

6.2 Axis Handles

All C-Motion procedures operate on an *axis handle,* which represents a single motor control axis of a motion processor. C-Motion axis handles are objects of type **PMDAxisHandle**, they should be allocated by the caller of the initialization procedure, and must not be overwritten until closed.

Axis handles may be initialized in several ways, depending on the kind of motion processor with which they are associated:

1. A “Magellan attached” device, for example a non-PRP ION or non-CME prodigy card, accessed either from a host or from a C-Motion Engine: In this case a peripheral connected to the Magellan is opened, a device handle initializes using the PMDMPDeviceOpen procedure and the **PMDAxisOpen** procedure is used to initialize an axis handle.

2. A PRP device motion processor, accessed from a host PC. In this case a peripheral connected to a PRP device must be opened, a device handle initialized using the **PMDRPDeviceOpen** procedure, and an axis handle initialized using the **PMDAxisOpen** procedure.
3. A PRP device motion processor, accessed from the C-Motion engine of the same device. In this case the **PMDAxisOpen** call is used with a null pointer as the device argument.

The following code examples illustrate the three options:

```
/* Case 1:  Magellan attached device. */

void testAxis(void)
{
   PMDPeriphHandle periph;
   PMDDeviceHandle device;
   PMDAxisHandle axis1;
   PMDresult result;

   /*
     Open a peripheral connection to a Magellan processor connected
     to a CAN bus.  CAN_TX and CAN_RX are CAN identifiers used for sending
     commands to and receiving responses from the processor, CAN_EVENT is a CAN
     identifier used for receiving asynchronous event notification.

     The device argument is NULL, because no remote device is used.
   */
   result = PMDPeriphOpenCAN(&periph, 0, CAN_TX, CAN_RX, CAN_EVENT);

   result = PMDMPDeviceOpen(&device, &periph);

   result = PMDAxisOpen(&axis1, &device, PMDAxis1);

   /* An example operation - soft reset the motion processor. */
   result = PMDReset(&axis1);

   /*
    Because the handles are allocated on the stack they
    closed before returning.
   */
   result = PMDAxisClose(&axis1);
   result = PMDDeviceClose(&device);
   result = PMDPeriphClose(&periph);
}

/* Case 2:  External PRP device Magellan motion processor. */

void testAxis(void)
{
   PMDDeviceHandle dev;
   PMDPeriphHandle periph;
   PMDAxisHandle axis1;
```

```
    PMDresult result;

    /*
       Open a peripheral connection to a PRP device using TCP/IP.
       IP_ADDRESS is the IP address of the PRP device, CMD_PORT
       is the TCP port used for sending commands to and
       receiving responses from the card, EVENT_PORT is the TCP port
       used for receiving asynchronous events.

    */
    result = PMDPeriphOpenTCP(&periph, 0, IP_ADDRESS, CMD_PORT, EVENT_PORT);

    /* Open a connection to the device on the peripheral. */
    result = PMDMPDeviceOpen(&dev, &periph);

    /* Now open a connection to the motion processor on the device. */
    result = PMDAxisOpen(&axis1, &dev, PMDAxis1);

    /* An example operation - soft reset the motion processor. */
    result = PMDReset(&axis1);

    /*
      Because the handles are allocated on the stack they
      closed before returning.
    */
      result = PMDAxisClose(&axis1);
    result = PMDDeviceClose(&dev);
    result = PMDPeriphClose(&periph);
}

/* Case 3: PRP device Magellan motion processor accessed from the CME. */

void testAxis(void)
{
    PMDAxisHandle axis1;
    PMDresult result;

    /* This is the simplest case:  pass NULL for the device argument to indicate
       the local device. */
    result = PMDAxisOpen(&axis1, 0, PMDAxis1);

    /* An example operation - soft reset the motion processor. */
   result = PMDReset(&axis1);

    /*
      Because the handles are allocated on the stack they
      are closed before returning.
    */
    result = PMDAxisClose(&axis1);
```

```
        }
```

This page intentionally left blank.

7. VB-Motion

VB-Motion is the interface from Microsoft Visual Basic .NET to the PMD C-Motion library for control of Magellan Motion Processors, which is documented in the *Magellan Motion Processor Programmer's Command Reference*. The Visual Basic interface documented in that manual is similar to but not identical to that used for PRP devices. Basic language programming is supported only for Microsoft Windows hosts, C-Motion Engine programming must be done in the C language.

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.

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 PRP device control.

Both debug and release versions of C-Motion.dll are provided in directories CMESDK\HostCode\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 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\VisualBasic.

Please note that the version of VB-Motion documented in the *Magellan Motion Processor Programmer's Command Reference,* while similar in spirit to the one described here, is not the same. In particular the class structure and the method of instantiating an axis object is different, and Sub methods containing the word "Get" or "Set" are not transposed. The Visual Basic interface described here does not use the Component Object Model (COM), but is instead supplied as Visual Basic source code.

The VB-Motion described here uses C-Motion 4.x. See Section 6.1, C-Motion Versions, for a discussion of the two threads of C-Motion development.

7.1 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**, **PMDPeripheralMultiDrop**, **PMDPeripheralCME**, **PMDPeripheralPCI**, and **PMDPeripheralTCP**.

Each class takes care of allocating and freeing the memory used for the "handle" structures used in the C language interface. The first pointer argument to, for example, a **PMDPeriphHandle** in a C language procedure call is not needed because a method call for a particular **PMDPe++ripheral** object is used instead, and each object manages its own **PMDPeriphHandle**.

The “Open” procedures used in the C language interface are replaced in Visual Basic with constructor methods that take the same arguments in the same order, with the exception that the first pointer argument is not needed. “Close” methods are provided that call the C language “Close” procedures, however these procedures may also be called automatically as part of the finalization process when objects are garbage collected.

The following example demonstrates how to open a peripheral connection to a PRP device accessible by TCP/IP, and to access the resources of that device.

```
Public Class Examples
    Public Sub Example1()

        ' Allocate and open a peripheral connection to a PRP device using TCP/IP.
        ' Note that the arguments for the PMDPeripheralTCP object are the same as for the
        ' C language call PMDPeriphOpenTCP, except that the first argument for the peripheral
        ' struct pointer and the second argument for the device are not used.
        ' The standard .NET class for IP addresses is used instead of a numeric IP address.
        ' DEFAULT_ETHERNET_PORT is a constant defined in PMDLibrary.vb for the default
        ' TCP port used for commands by the PRP device.
        ' 1000 is a timeout value in milliseconds.
        Dim periph As New PMDPeripheralTCP(System.Net.IPAddress.Parse("192.168.0.27"), _
                                    DEFAULT_ETHERNET_PORT, _
                                    1000)

        ' Now allocate and connect a device object using the newly opened peripheral.
        ' Instead of using two different names the second argument specifies whether a
        ' PRP device or attached Magellan device is expected.
        Dim DevCME As New PMDDevice(periph, PMDDeviceType.ResourceProtocol)

        ' Once the PRP device is open we can obtain an axis object, which may be used
        ' for any C-Motion commands. Notice that the enumerated value used to specify the axis is
        ' called "Axis1" instead of "PMDAxis1" because the enumeration name already includes
        ‘ the “PMD” prefix.
        Dim axis1 As New PMDAxis(DevCME, PMDAxisNumber.Axis1)

        ' Open an object representing the dual-ported RAM on the PRP device.
        ' Currently only 32 bit memory devices are supported, but other devices may be supported
        ' in the future.
        Dim mem As New PMDMemory(DevCME, PMDDataSize.Size32Bit)

        ' Start the user program on the C-Motion Engine.
        DevCME.TaskStart()

        ' 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.
         Dim pos As Int32
         pos = axis1.ActualPosition

        ' The following line sets the actual position of the axis to zero.
        axis1.ActualPosition = 0
        ' Properties may accept parameters, for example the CurrentLoop parameter is used to set
         ' control gains for the current loops, and takes two parameters. This example sets
         ' the proportional gain for phaseA to 1000
        axis1.CurrentLoop(PMDCurrentLoopNumber.PhaseA,  _
                              PMDCurrentLoopParameter.ProportionalGain) = 1000

        ' C-Motion procedures returning multiple values become Sub methods, and return their
        ' values using ByRef parameters. The "Get" and "Set" parts of the names are the same as
        ' in the C language binding.
        Dim MPmajor, MPminor, NumberAxes, special, custom, family As UInt16
        Dim MotorType As PMDMotorTypeVersion
        axis1.GetVersion(family, MotorType, NumberAxes, special, custom, MPmajor, MPminor)

        ' C library procedures that accept pointers become methods operating on VB arrays.
        ' This example reads 100 values from dual-ported RAM.
        Dim memvals(0 To 100) As UInt32
        mem.read(memvals, 100, memvals.Length())

        ' If the objects opened here are not explicitly closed they will be closed by the
        ' garbage collector.
    End Sub
End Class
```

Several general points about the translation from C to Visual Basic are shown in the example:

- Argument type and order are the same, except that the initial “handle” pointer argument is not needed. The null device pointer used to indicate that a peripheral is opened on the local device is also not needed.
- “Get/Set” procedures returning a single argument become object properties, with parameters if needed. The property name does not contain “Get” or “Set”, or the “PMD” prefix.
- Procedures returning or setting multiple values are implemented as Sub methods, returning values via ByRef parameters. “Get” or “Set” is retained in the names, but the “PMD” prefix is not.
- Enumerated value names do not use the “PMD” prefix, but the enumeration names do.
- Procedures reading or writing array data through C pointers instead take Visual Basic arrays of the appropriate type.

7.2 Using A Magellan Attached Device

The following example illustrates how to obtain an object connected to a Magellan attached device, in this case connected to a serial port.

```
Public Class Examples
    Public Sub Example2()
        Dim periph As PMDPeripheral
        Dim MPDev As PMDDevice
        Dim axis2 As PMDAxis
        ' Open the connection on COM1, using appropriate serial port parameters
        periph = New PMDPeripheralCOM(1, PMDSerialBaud.Baud57600, _
                                      PMDSerialParity.None, PMDSerialStopBits.Bits1)

        ' Obtain a device object using the peripheral. Device type MotionProcessor
        ‘ means “Magellan attached”.
        MPDev = New PMDDevice(periph, PMDDeviceType.MotionProcessor)

        ' Finally instantiate an axis object, specifying the desired axis number 2.
        axis2 = New PMDAxis(MPDev, PMDAxisNumber.Axis2)

        ' Example VB-Motion operation: Get the event status
        Dim status As UInt16
        status = axis2.EventStatus
    End Sub
End Class
```

7.3 Error Handling

Almost all of the PMD C language library procedures return an error code to indicate success or failure. The Visual Basic versions of these procedures instead throw an exception if the wrapped DLL procedures return an error code. The exception message will contain the error number and a short description of the error. The Data member of the exception will contain the error number as an enumeration of type **PMDresult**, associated with the key “PMDresult”, so that structured exception handling may be used

to appropriately handle errors.

The following example commands a PRP device to reset, and then ignores the expected error return on the next command:

```
dev.Reset()
Try
    Dim major, minor As UInt32
    dev.Version(major, minor)
Catch ex As Exception When ex.Data("PMDresult").Equals(PMDresult.ERR_RP_Reset)
    ' Ignore the expected error
 End Try
```

Any errors that are not caught will cause the application to display a popup window displaying an error message, including the error number and description, and a stack trace with file names and line numbers. The popup window allows a user to continue, ignoring the error, or to abort the application.

While popup windows are useful for debugging, any application controlling motors should be designed to recover gracefully and safely from any foreseeable error condition, and it is recommended to use Try blocks liberally to make applications more robust.

Index

A

action reference 29
actions 22
addresses 22
axis handles 129

C

C language library procedure 77
CANbus transport 28
C-Motion 123
    Axis handles 129
    Engine 123
    Engine macros 78
    Engine Procedures 123
    Engine programming 124
    libraries 123
    versions 129
console 127

D

data types 77

E

event notification packet 24
exceptions, C-Motion 127

N

naming conventions 77

O

outgoing PRP packet 23
overview, devices 7

P

packet
    event notification 24
    outgoing PRP 23
    response 23
    structure 23
PCI bus transport 27
PMD library procedures 79
PRP
    action reference 29
    actions 22
    addresses 22
    CANbus transport 28
    devices, overview 7
    outgoing packet 23
    packet structure 23
    PCI bus transport 27
    resources 21
    response packets 23
    serial transport 26
    sub-actions 22
    TCP/IP transport 27
    transport layers 25
    tutorial 9
PRP Tutorial 9
    actions 11
    addressing 9
    auto-assigned addresses 14
    CANbus 19
    communications formats 18
    communications ports 11
    description 9
    Ethernet 19
    magellan-attached device 16
    on-card resources 15
    peripheral connections 12
    resources 11
    serial 18

R

resources, PRP 21
response packets 23
return values 78

S

scope 7
serial transport 26
sub-actions 22

T
TCP/IP transport 27
transport layers 25

U
user packets 127

V
VB-Motion 133
    error handling 136
    Magellan-atttched device 135
    Visual Basic, classes 133
versions, C-Motion 129

**For additional information, or for technical assistance,
please contact PMD at (978) 266-1210.**

You may also e-mail your request to support@pmdcorp.com

Visit our website at http://www.pmdcorp.com

PMD

Performance Motion Devices
80 Central Street
Boxborough, MA 01719