```
code for executing a profile and tracin
aptured in this example could be used for tuning th
trace buffer wrap mode to a one time trace
raceMode (hAxis1, PMDTraceOneTime);
t the processor variables that we want to capture
tTraceVariable (hAxis1, PMDTraceVariable1, PMDAx
etTraceVariable (hAxis1, PMDTraceVariable2, PMDAxi
SetTraceVariable (hAxis1, PMDTraceVariable3, PMDAxis
// set the trace to begin when we issue the next update command
SetTraceStart (hAxis1, PMDTraceConditionNextUpdate);
// set the trace to stop when the MotionComplete event occurs
SetTraceStop (hAxis1, PMDTraceConditionEventStatus,
  PMDEventMotionCompleteBit, PMDTraceStateHigh);
etProfileMode (hAxis1, PMDTrapezoidalProfile);
et the profile parameters
Position(hAxis1, 200000);
elocity(hAxis1, 0x200000);
eleration(hAxis1, 0x1000);
eration(hAxis1, 0x1000);
```

C-Motion PRP

Programming Reference

Revision 1.2 / October 2023

Performance Motion Devices, Inc.

80 Central Street, Boxborough, MA 01719

www.pmdcorp.com

NOTICE

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

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

Warranty

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

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

Unauthorized Applications

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

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

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

Disclaimer

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

Patents

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

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

Related Documents

Magellan Motion Control IC User Guide

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

C-Motion Magellan Programming Reference

Descriptions of all C-Motion Magellan Motion Control IC commands, with coding syntax and examples, listed alphabetically for quick reference.

C-Motion Engine Development Tools Manual

Describes the C-Motion Engine Development Tools that allow user application code to be created and compiled on a host PC, then downloaded, executed and monitored on a CME device C-Motion Engine module.

ION/CME 500 Digital Drive User Manual

Complete description of the ION/CME 500 Digital Drive including getting started section, operational overview, detailed connector information, and complete electrical and mechanical specifications..

Prodigy/CME Machine-Controller User Guide

Complete description of the ION/CME 500 Digital Drive including getting started section, operational overview, detailed connector information, and complete electrical and mechanical specifications..

Table of Contents

This page intentionally left blank.

1. Introduction

In This Chapter

- Introduction
- PMD Products and C-Motion Version
- Overview of C-Motion PRP

1.1 Introduction

This manual documents C-Motion PRP, which is a software library used to control and monitor various PMD motion control products. PRP stands for PMD Resource Access Protocol, which is the protocol used to communicate with these devices.

There are two other C-Motion versions; C-Motion Magellan and C-Motion PRP II. All of these software systems are available in separate SDKs as detailed below:

- **C-Motion Magellan SDK** – an SDK (Software Developer Kit) for creating motion applications using the C/C++ programming language for PMD products that utilize a direct Magellan or Juno formatted protocol.
- **C-Motion PRP SDK** – an SDK for creating PC and downloadable user code for systems utilizing either a PRP (PMD Resource Access Protocol) protocol device or a Magellan/Juno protocol device. C-Motion PRP is also used in motion applications that will use the .NET (C#, VB) programming languages.
- **C-Motion PRP II SDK** – This SDK is similar to C-Motion PRP but is used with ION/CME N-Series ION Digital Drives. Compared to standard C-Motion PRP, C-Motion PRP II supports additional features such as multi-tasking, mailboxes, mutexes, and enhanced event management.

For detailed information on Magellan/Juno protocol C-Motion refer to the *C-Motion Magellan Programming Reference*. For detailed information on C-Motion PRP II refer to the *C-Motion PRP II Programming Reference.*

1.2 PMD Products and C-Motion Version

The following table shows the C-Motion versions that can be used with each PMD product family:

Product Family	Compatible C-Motion Versions
Magellan ICs	C-Motion Magellan, C-Motion PRP*
Juno ICs	C-Motion Magellan, C-Motion PRP*
ION/CME N-Series	C-Motion PRP II
ION 500	C-Motion Magellan, C-Motion PRP*
ION/CME 500	C-Motion PRP
ION 3000	C-Motion Magellan, C-Motion PRP*
Prodigy PC/104	C-Motion Magellan, C-Motion PRP*

Prodigy/CME PC/104	C-Motion PRP
Prodigy/CME Stand-Alone	C-Motion PRP
Prodigy/CME Machine-Controller	C-Motion PRP

*C-Motion PRP typically only used for .NET support, or if a mix of Magellan/Juno protocol and PRP protocol devices are attached.

1.3 Overview of C-Motion PRP

C-Motion is PMD's C-language based motion control programming system. It is provided in source code form for easy integration on a wide variety of platforms. Its primary purpose is to provide a C-language API to interface with, and access the resources of, PMD's motion control products.

All PMD products utilize packet-based protocols for communication, so a primary purpose of C-Motion is to translate the information contained in C-language function calls to the proper packet format. This allows C-Motion application developers to avoid having to learn the low level communication formats required by each PMD product.

Within the full PMD product set there are two different packet protocols used. A protocol known as the Magellan/Juno protocol is used when directly interfacing with PMD Magellan ICs or Juno ICs. PRP (PMD Resource Access Protocol) is the protocol used with products such as ION/CME Digital Drives and Prodigy/CME boards.

Not all C-Motion function calls are translated into packets that will be sent, or received, by a PMD product. Especially for C-Motion PRP or C-Motion PRP II libraries, many function calls are used to manage application execution, memory resources, tasks, or to access resources located within the same device executing the C-Motion engine user code.

1.3.1 Resource Access Virtualization

In addition to handling the details of packet protocol conversion, another important feature of C-Motion is its support for virtualization of resource access.

Whether accessing a Magellan Motion Control IC, a memory block, a digital I/O port, or a CANbus peripheral port, C-Motion calls accept a handle which provides access to that resource independent of its location on a network or even PMD product type.

To instantiate a particular resource handle C-Motion calls are used to establish needed access information. It is this handle that is then provided to downstream C-Motion calls which command, or query, that resource. We will discuss the specifics of initializing access information in more detail later, but what is important about access virtualization is that it makes it easy to re-use previously written code for new machine control projects, or to transport code from prototyping setups to custom-designed production boards.

1.3.2 C-Motion Code Execution

A special and unique capability of the C-Motion PRP system is that it allows application code sequences to be run either from an external host (such as a PC) or from the C-Motion Engine on the device. This is convenient for code development, which is often easier and faster when located on the PC.

When operating on a host PC the C-Motion PRP system converts C-Motion calls to PRP protocol packets and sends them through the network interface to the device. This same C-Motion application code, when re-compiled for operation on the target device's C-Motion Engine (sometimes called CME for short) no longer sends packets in PRP format but instead makes the conversions needed to access the on-device resources from the CME, using the device's internal high speed communication bus.

1.3.3 Communication Networks

Another unique and powerful feature of the C-Motion PRP system is that it allows layered networks to be created. For example if a host PC talks directly to a Prodigy/CME Machine-Controller board via an Ethernet connection this board can in turn have a network of ION/CME units attached through its CAN network interface.

PRP allows both the resources on the Prodigy board and the 'sub network' ION/CME resources to be seamlessly addressed from the PC. Built into the PRP resource accessing scheme is the capability for devices to act as network gateways, directly processing messages intended for local resources, and passing on messages intended for resources connected by network to the local device.

From the perspective of the C-Motion user code running on the PC access to all resources is automatic. To achieve this, as before, once the location of the devices and resources of the PRP network is established through C-Motion initialization calls, subsequent calls use just a C-language handle, whether the resources is directly-connected, or connected through a network.

In the next chapter we will expand on all of these concepts and give examples of how C-Motion PRP II is used to achieve various common control functions.

This page intentionally left blank.

2. PMD Resource Access Protocol (PRP)

In This Chapter

- Introduction
- PRP Resources
- PRP Actions and Sub-Actions
- PRP Addresses
- PRP Packet Structure
- Using PRP

2.1 Introduction

Access to Prodigy/CME boards, ION/CME Drives, and Ethernet-capable ION drives is provided by a protocol called PMD Resource Access Protocol (PRP). PRP may be transmitted via serial, CAN, Ethernet TCP/IP, or SPI (Serial Peripheral Interface). PRP is both a protocol which can be transmitted across various connection interfaces and an architecture for how resources on PRP devices are accessed. A complete understanding of C-Motion PRP therefore requires an understanding of PRP.

PRP device functions are organized into *resources*; resources 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 device or connected to the device.

A basic communication to a PRP device consists of a 16 bit PRP header and for some communications a message body. The message body, if present, contains data associated with the specified PRP action. The header contains various information used to process the PRP messages including identifiers for the resource type, action type, and resource address. After a PRP communication is sent to a device, a return communication is sent by the PRP device which consists of a response 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.

PRP is a master/slave system. The host functions as the master and initiates communication sequences which the connected device must respond to. The connected device can not initiate messages on its own within the PRP protocol. Note however that some PRP-supported networks, in particular CAN and Ethernet, allow one or more non-PRP protocol connections to be established to support asynchronous communication from the attached device to the host.

In the sections below more information is provided on each of these PRP constructs.

2.2 PRP Resources

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

The following table summarizes the various resource types and their numeric codes as specified in the header.

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

2.3 PRP Actions and Sub-Actions

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 serial, CAN, Ethernet, or SPI, *Read* and *Write*, which are used to access memory-type devices, and *Set* and *Get*, which are used to load or read parameters.

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 command,* another 8 bit qualifier of the action to take.

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

Name	Value	Meaning
NOP	0	No operation
Reset	1	Perform a reset
Command	2	Motion Processor and miscellaneous actions
Open	3	Open an addressable resource
Close	4	Close a remote resource
Send	5	Send data to a stream-like resource
Receive	6	Receive data from a stream-like resource
Write	7	Write data to an indexed resource
Read	8	Read data from an indexed resource
Set	9	Change a setting or operating state
Get	10	Get a setting or operating state
Clear	11	Erases the memory resources

2.4 PRP Addresses

Every resource accessible via PRP is identified by a numeric address. Addresses for Memory, Motion Processor, and C-Motion Engine resources local to a PRP device are fixed numbers. Refer to the user manual for the C-Motion PRP-based product you are using for a detailed list. Addresses for Peripheral resources and resources on remote PRP devices, that is devices not directly connected to the host, are obtained by PRP actions and are automatically assigned. For more information on automatically assigned see Section 2.6.2, Automatically Assigned Addresses and Peripherals

While these automatically assigned 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.

2.5 PRP Packet Structure

2.5.1 Outgoing PRP Packet

The core of the PMD Resource Access Protocol is a header that accompanies all PRP communications. The figure below 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.

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…*								

PRP outgoing packet header 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 communications being sent out by the host, this 2 bit field should contain the value 2.

Action - This 4 bit field contains an action identifier that is used to process PRP messages. See Section 2.3, PRP Actions and Sub-Actions, for a summary of the PRP actions supported by PRP.

Resource - This 3 bit field encodes the specific resource type being addressed. See the table in Section 2.2, PRP Resources, for the summary of resources supported by PRP.

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. 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.

2.5.2 PRP Response Packet

When an outgoing PRP packet is received by the device it responds with a response packet, which consists of 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 is used to determine the length.

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

PRP Success Response Packet

header *byte 0*	version (1)		status (0)		reserved			
	7	6	5	4	3	2	1	0
body *byte 0…*								
	7	6	5	4	3	2	1	0

PRP Failure Response Packet

header *byte 0*	version (1)		status (1)		reserved			
	7	6	5	4	3	2	1	0
	error *byte 0*							
	7	6	5	4	3	2	1	0
	error *byte 1*							
	7	6	5	4	3	2	1	0

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

The bits marked reserved must have a value of zero.

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 some values that the error code may take. (See the C-Motion PMDecode.h source file for all the possible values.) 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
Checksum	0x200E	Bad packet checksum value
Magellan error codes	1 – 35	Magellan Motion Processor error codes, documented in the *Magellan Motion Control IC User Guide.*

2.6 Using PRP

In the next few sections we will provide examples of important PRP concepts including how to access resources, how to use automatically assigned addresses, and more.

Beyond these examples here is a list of additional useful C-Motion PRP II resources contained in this manual:

- Section 3.7, Alphabetical C-Motion API Reference, provides detailed information on the C-Motion PRP API, listed alphabetically
- Section 4.2, Action Table - Alphabetical Order, provides detailed information including packet format for all PRP Actions, listed alphabetically

- Section 3.7, Alphabetical C-Motion API Reference, provides an alphabetically listed table of the C-Motion PRP II API and its corresponding PRP Actions
- Section 4.1, Action Table - Code Order, provides the same information but in reverse, a table of PRP Actions and the corresponding C-Motion PRP API
- Appendix A, *PRP Transport*, provides detailed information on the format and process for transporting PRP on Serial, CAN, Ethernet, or SPI

2.6.1 Device Access Basics

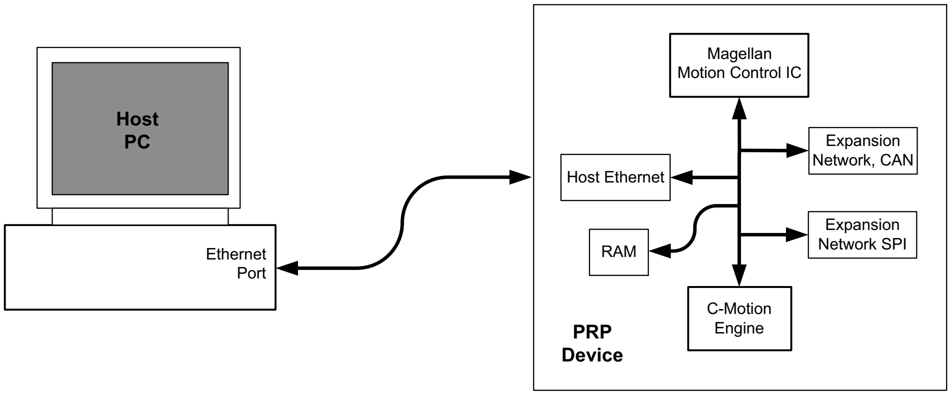


Figure 2-1: Host PC Connected to PRP Device via Ethernet TCP

Accessing resources on PRP devices is straightforward using the C-Motion PRP system. To illustrate this we will begin by showing the C-Motion commands used to achieve this. We will then illustrate how this same function is achieved via PRP-formatted packets.

Example 1: A Host Controller is connected to a PRP device via Ethernet/TCP and sets the position of Axis #3 of the PRP device's onboard Magellan Motion Control IC to a value of 0x123456.

Example in C-Motion

The first step will be to create an Ethernet/TCP peripheral connection and associated C-language handle on the host PC. Then we use this peripheral handle to create a handle to access the Ethernet-connected PRP device. Finally, using this device handle we will open an Axis handle which is used to access all Magellan Motion Control IC commands.

```
PMDPeriphOpenTCP()*          // Open and get access handle for TCP Peripheral on Host PC
PMDRPDeviceOpen()            // Open PRP-based device via this peripheral connection
PMDAxisOpen()                // Get Magellan Axis handle at axis #3 using PRP device handle
PMDSetPosition()             // Send SetPosition 0x123456 from PC to Magellan IC
```

**For clarity the content of these example C calls such as handles and other initialization information will not be shown. For complete C-Motion coding examples refer to CMESDK\HostCode\Examples located on the C-Motion PRP SDK.*

Note that once we have a handle set up we may use it to access the associated resource without re-opening that resource. For example in the above sequence if we want to also set the motion control IC's motion velocity, we would just add a **PMDSetVelocity()** call to the above sequence using the same axis handle as was used to set the position.

Example in PRP

The above example in PRP format looks very different. There are two reasons for this, one of which is that the mnemonic format for PRP packets is different than C language calls. The general PRP packet mnemonic format is:

<Resource ID> <Address> <Action ID> <Message content>

The other reason is that none of the C-Motion initialization calls which create virtual resource access through handles are relevant. So the PRP sequence is a single packet which is sent to the *MotionProcessor* resource, and has an action type of *Command*.

From the table in Section 2.2, PRP Resources, through Section 2.4, PRP Addresses, to communicate with the onboard Magellan Motion Control IC, a PRP message is sent to Resource ID 2 (corresponding to the *MotionProcessor* resource), address 0 (corresponding to the PRP device's onboard Magellan address), and 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.

In PRP mnemonics here is this command:

MotionProcessor, Addr 0, Command, 0x0210, 0x0012, 0x3456

Upon processing of this command by the device, the host would receive a PRP response 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 same Host Controller wants to read the 32 bit word value of address 0x100 of the PRP device's RAM

Example in C-Motion

Here we will send a **PMDMemoryRead()** call to retrieve the memory. From the previous Example #1 sequence we will assume the first two initializations have already been made and now execute the additional needed calls:

```
PMDMemoryOpen32()          // takes the device handle and creates a memory resource handle
PMDMemoryRead()            // takes the memory resource handle and returns the requested data
```

Example in PRP

The ID for a *Memory* resource type is 3, and the ID for a *Read* action is 7. The message body contains a sub-action of 0 specifying a 32 bit word read followed by a 0x100 which specifies the address of the desired memory read. Upon successfully processing this command, the host would receive the 32 bit contents of memory location 0x100 in the message body.

So in PRP mnemonics here is this outgoing command:

Memory, Addr 0, Read, 0, 0x100

Note that the PRP *Command* message sent to the Magellan Motion Control IC did not use a sub-action code in the message body, while the *Read* command sent to the 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 4, *PRP Action Reference,* provides exact message body information for each PRP action and (if applicable) sub-action.

2.6.2 Automatically Assigned Addresses and Peripherals

The above examples illustrate how C-Motion PRP is used to gain basic access to on-device resources. In these examples the address of the resource being commanded or queried were local to the device, and therefore had a fixed numerical value.

In the PRP system however there are instances where the device or resource address is not fixed and is assigned dynamically. These occurs in particular when addressing the *Peripheral* resource.

PRP devices support up to three different network connection types; Serial, CAN, and Ethernet. These communication resources are represented in PRP by a construct called a peripheral connection. A peripheral is a resource (resource ID: 4), and is used to send and receive messages to network connections.

Obtaining access to an on-device serial, CAN, Ethernet, or SPI port is accomplished via the PRP *Open* action. This action opens a peripheral by specifying a sub-action of *PeriphSerial*, *PeriphCAN*, *PeriphTCP*, or *PeriphUDP*. The corresponding C-Motion commands are **PMDPeriphOpenCOM()**, **PMDPeriphOpenCAN()**, **PMDPeriphOpenTCP()**, and **PMDPeriphOpenUDP()**.

The addresses of these Peripheral resources are not fixed. Each newly opened peripheral connection receives an automatically assigned address within the PRP response message body. The device that requests the peripheral open connection must record that provided address for future use, and it is this address that is used in subsequent PRP messages to that peripheral connection.

Note that automatically assigned addresses generally increment by one each time they are assigned, however this should not be assumed.

Opening a new peripheral opens a connection between a PRP device and a specific remote device. It does not open the overall network port. For example if a PRP device has a CAN network with 4 attached devices (each at seperate CAN network addresses), four separate open peripheral function calls must be made, each opening a one-to-one connection between the PRP device and a specific network-attached device.

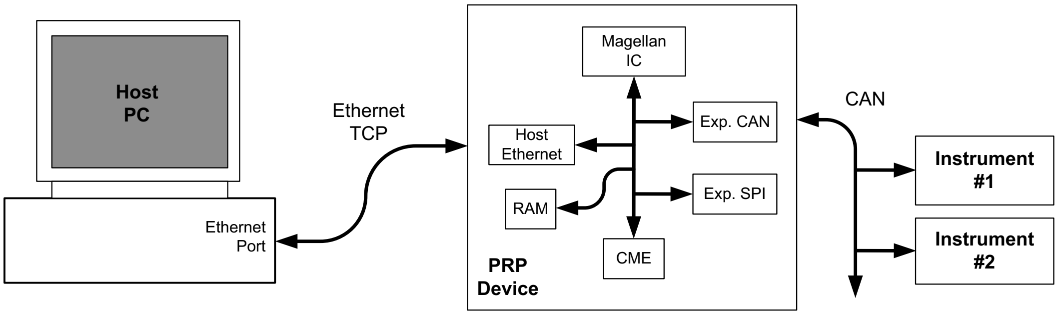


Figure 2-2: Host PC Connected to PRP Device connected to Instruments via CAN Network

Example 1

Figure 2-2 **shows a network configuration. A Host PC is connected via Ethernet TCP to a PRP device, which in turn is connected via a CANFD network to two scientific instruments. The host controller needs to initiate, send and receive a message to/from the CAN-connected instrument.**

Example in C-Motion

The first two steps provide general Ethernet access from the PC to the PRP device, and are the same as from our previous examples.

```
PMDPeriphOpenTCP()*          // Open TCP Peripheral connection on Host PC
PMDRPDeviceOpen()            // Open PRP-based device connection
```

Next we use the device handle created using the open PRP device call to access the Ethernet-connected PRP device and open CANFD peripherals to each instrument. Using this peripheral handle we then send and receive a message:

```
PMDPeriphOpenCAN()           // Open CAN Peripheral connection #1
PMDPeriphOpenCAN()           // Open CAN Peripheral connection #2
PMDPeriphSend()              // Send a message to the #1 peripheral connection
PMDPeriphReceive()           // Receive a message from #1
PMDPeriphSend()              // Send a message to the #2 peripheral connection
PMDPeriphReceive()           // Receive a message from #2
```

**For clarity the contents of the C calls such as handles and other initialization/parameter information is not shown.*

Example in PRP

As in the examples from the previous section there are no PRP transactions to set up resource or peripheral access handles. So the first step is to open a CANFD peripheral connection on the PRP Device.

Device, Addr 0, Open, PeriphCANFD, <CANFD Parameters for #1>
Device, Addr 0, Open, PeriphCANFD, <CANFD Parameters for #2>
Peripheral, <Assigned Address for #1>, Send, <Message>
Peripheral, <Assigned Address for #1>, Receive, <Message>
Peripheral, <Assigned Address for #2>, Send, <Message>
Peripheral, <Assigned Address for #2>, Receive, <Message>

In the return message body of the first transaction above the automatically assigned address of the opened CANFD peripheral is provided, and this address is used for the subsequent *Send* and *Receive* actions. <CANFD Parameters> here denotes that the message body of the outgoing communication contains formatted information indicating the Node ID.

Upon processing the peripheral receive command the PRP device will wait for a CANFD message to be received. A timeout value can be provided so that the length of this wait period can be limited. Once the message is received the PRP response message contains the received CANFD message.

2.6.3 RS232 & RS485 Peripherals

Most PMD products support both RS232 and RS485 serial communications, although specifying that a serial port should operate as a RS485 network reduces the number of serial ports available. For example PMD's N-Series ION Drive supports separate Serial1 and Serial2 point-to-point RS232 connections but just Serial1 when configured for multidrop RS485 operation.

Opening a point-to-point serial connection is straightforward and uses the C-Motion call **PMDPeriphOpenCOM()**. In the argument list the port is specified (Serial1, Serial2, or Serial3) along with other parameters such as baud rate, parity, etc.

In PRP protocol this is:

Device, Addr, Open PeriphSerial, <Serial Parameters>

Opening a multi drop RS485 connection however requires two calls, the first to open a serial peripheral connection, and then separate calls for each RS485 connection that is to be created. This second peripheral open uses what is called a multi drop peripheral type. Here is what this call sequence looks like via C-Motion, showing how devices at two separate RS485 network addresses are connected to.

```
PMDPeriphOpenCOM()            // open serial port peripheral, creating periph handle
PMDPeriphOpenMultiDrop()      // open multi drop peripheral connection # 1 using
                              // above serial periph handle. Resultant peripheral handle
                              // now represents the RS485 connection to the device at the
                              // first RS485 address
PMDPeriphOpenMultiDrop()      // open multi drop peripheral connection # 2 using
                              // original serial periph handle. Resultant peripheral handle
                              // now represents the RS485 connection to the device at the
                              // second RS485 address
```

Here is the same sequence in PRP mnemonics:

Device, Addr, Open, PeriphSerial, <Serial Parameters>
Periph, <Assigned Addr>, Open PeriphMultiDrop, <RS485 connection parameters for node #1>
Periph, <Assigned Addr>, Open PeriphMultiDrop, <RS485 connection parameters for node #2>

After these sequences there are two multidrop peripherals which can then be used for communications to and from each connection via standard peripheral *Send* or *Receive* commands.

2.6.4 Remote Attached Devices

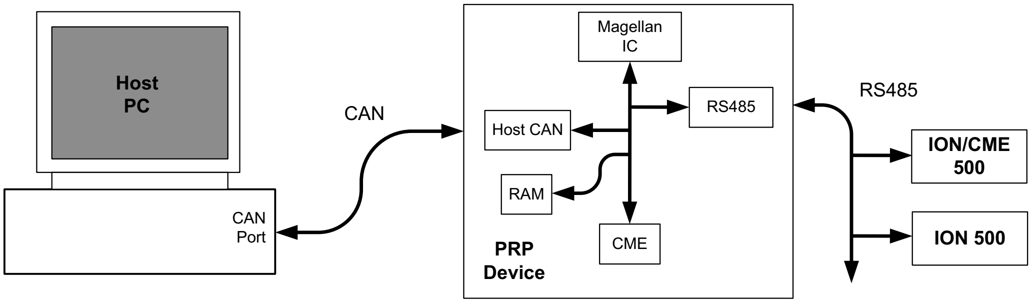


Figure 2-3: Host PC Connected to PRP Device connected to ION/CME and ION 500 via RS485 Network

Before closing our discussion of peripheral connections there is one more especially useful configuration to discuss. In Figure 2-3 a host PC connects to a PRP device which in turns has additional devices connected to it via another network. These additional devices, from the perspective of the PC, are referred to as remote attached devices. With PRP, creating ‘bridged’ networks like this is not difficult, as this example shows.

Example

A Host PC is connected via CAN to a PRP device, which in turn is connected via RS485 to two devices; an ION/CME 500 (#1) and an ION 500 (#2). The host controller needs to set a destination position, and send a GetVersion command to both of the remote RS485 connected ION Drives.

Example in C-Motion

The first two steps provide general CAN access from the PC to the PRP device, and are similar to our previous examples other than the switch from Ethernet to CAN.

```
PMDPeriphOpenCAN()          // Open CAN Peripheral connection on Host PC
PMDRPDeviceOpen()           // Open PRP-based device connection
```

Next we will open a serial peripheral connection so that we can create two RS485 connections, one to each device.

```
PMDPeriphOpenCOM()              // Open Serial peripheral connection
PMDPeriphOpenMultiDrop()        // Open multi drop peripheral connection # 1
PMDPeriphOpenMultiDrop()        // Open multi drop peripheral connection # 2
```

Next we will create device connections via each of these peripherals. This accomplished via either an OpenDevicePRP call (for PRP protocol devices) or an OpenDeviceMP (for Magellan/Juno format devices). In this example the #1 device is an ION/CME and therefore a PRP device, while the #2 device is an ION 500 and therefore a Magellan/Juno protocol device.

```
PMDRPDeviceOpen()           // Open PRP device connection for #1 ION (ION/CME 500)
PMDMPDeviceOpen()           // Open Magellan device connection for #2 ION (ION 500)
```

Finally we create access handles to the motion processor axes for each device and set the destination position command and query the unit version.

```
PMDAxisOpen()                       // Using handle for device #1 get Magellan axis handle
PMDAxisOpen()                       // Using handle for device #2 get Magellan axis handle
PMDSetPosition()                    // Set position to 0x123456 to Axis on device #1
PMDSetPosition()                    // Set position to 0x234567 to Axis on device #2
PMDGetVersion()                     // Query version of Magellan on device #1
PMDGetVersion()                     // Query version of Magellan on device #2
```

Example in PRP

Since we don’t need commands to create handles to access the Host PC-attached device, the first step is to open a serial peripheral connection, then we create two RS485 peripheral connections, first for device #1 and next for device #2

Device, Addr 0, Open, PeriphSerial, <Serial parameters>
Device, <assigned Addr>, Open, PeriphMultiDrop, <RS485 parameters for #1>
Device, <assigned Addr>, Open, PeriphMultiDrop, <RS485 parameters foir #2>

Next we will create device connections via each of the just-created RS485 peripheral addresses.

Periph, <assigned Addr>, Open, DevicePRP, <Parameters for PRP Device>
Periph, <assigned Addr>, Open, DeviceMP, <Parameters for MP Device>

Finally we send the desired **SetPosition** and **GetVersion** commands to each motion control IC.

MotionProcessor, <device Addr #1>, Command, <SetPosition 0x123456>
MotionProcessor, <device Addr #2>, Command, <SetPosition 0x234567>
MotionProcessor, <device Addr #1>, Command, <GetVersion>
MotionProcessor, <device Addr #2>, Command, <GetVersion>

Note that in the above PRP messages the commands sent to the motion processor resource are not sent as ASCII characters but rather in a packet protocol format. In the mnemonics they are shown in ASCII only for clarity. Magellan IC packet formats are detailed in the *C-Motion Magellan Programming Reference*.

2.6.5 Other Peripheral Types

As it turns out there are some peripheral types that do not strictly function as communication ports, but are still accessed as *Peripheral* resources. These peripheral types are listed in the table below. Note that some of these peripheral types, rather than using *Send* and *Receive* commands, use *Read* and *Write* commands to access their contents.

Peripheral Type (Sub Action Name)	Description
PeriphPRP	PRP Peripherals allow general purpose application-specific communications to occur through an already established PRP channel. This mechanism, often referred to as tunneling, can be convenient for “conversation constrained” network interfaces such as Serial or SPI.
PeriphPIO	Each PRP Device has a single PIO Peripheral which gives access to various bit or word encoded registers. These registers provide read or write access to the unit’s Digital I/O bits, analog inputs, encoder-related settings, and more.

3. PMD C-Motion API Reference

In This Chapter

3.1 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."

3.2 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, and are passed to library procedures by using a pointer argument. They 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 protocol, that is, a Prodigy/CME card or an ION/CME module; an MP device is a device that communicates using the Magellan/Juno 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 Control IC, 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 memory resource 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.

3.3 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 Section 2.5.2, PRP Response Packet, for more information.

3.4 C-Motion Engine

The C-Motion Engine is a special purpose computer included in PMD’s CME line of products, and connected by a high speed internal bus to the on-board Magellan Motion Processor, memory, 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.

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 most CME devices is approximately 7 kilobytes. The heap in N-Series ION devices is approximately 500k.

CME tasks can be aborted using PMDTaskAbort. Do not return from a CME task function.

3.4.1 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.

C-Motion Engine programs are compiled with the GNU C compiler (GCC) provided with the CME SDK. Each example contains a build.bat file that builds the appropriate example. The resulting binary file is then downloaded to the CME device via Pro-Motion or the command-line utility StoreUserCode.exe.

3.4.2 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 “PMDsys.h” file.

USER_CODE_VERSION (MAJOR, MINOR)
USER_CODE_TASK (myProgram)

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 "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 (myProgram)
{
…

   while (l) {
   // Handle task events
   }
   PMDTaskAbort(0);
}
```

3.5 Microsoft .NET Programming

3.5.1 Visual Basic Programming

The Visual Basic PMD Library is the interface from Microsoft Visual Basic .NET to the PMD C-Motion library for control of Magellan Motion Control ICs, which is documented in the *Magellan Motion Control IC Programming 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 or Magellan 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\DotNet.

3.5.2 Visual Basic Classes

The file PMDLibrary.vb defines a Visual Basic class for each of the opaque data types used in the PMD library: **PMDPeripheral**, **PMDDevice**, **PMDAxis**, and **PMDMemory**. **PMDPeripheral** is inherited by a set of derived classes for each peripheral type: **PMDPeripheralSerial**, **PMDPeripheralMultiDrop**, **PMDPeripheralPRP,** **PMDPeripheralCAN**, 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 **PMDPeripheral** 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 PMDDeviceOpenPeriphTCP, 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)

' 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)

' 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.

3.5.3 C# Programming

The C# language is very similar to the VB language. A C# PMD program uses the PMDLibrary.dll created by the ClassLibrary project located in CMESDK\HostCode\DotNet\ClassLibrary. An example C# PMD program can be found in CMESDK\HostCode\DotNet\CSTestApp.

3.5.4 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.

3.6 PMD Library Procedures

This section documents the PMD C language interface to the library procedures for programming a CME 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 a PRP action, or none. 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.

3.7 Alphabetical C-Motion API Reference

This page intentionally left blank.

PMDAxisOpen

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

C-Motion Engine	Host-Based

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

PMDDeviceReset

Arguments:

name	type
hDevice	pointer to PMDDeviceHandle

C language syntax:

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

Visual Basic Syntax:

```
device.Reset()
```

Description: **PMDDeviceReset** is used to reset the device. 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

PMDDeviceSetDefault

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

PMDDeviceGetVersion

C-Motion Engine | Host-Based

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

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

PMDMemoryClose

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

PMDMemoryRead

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

PMDMemoryWrite

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

PMDPeriphOpenCME

C-Motion Engine | **Host-Based**

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

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

PMDPeriphOpenISA

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

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

PMDPeriphOpenPCI

Host-Based

Arguments:

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.

PMDPeriphOpenPIO

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

Description:

PMDPeriphOpenPIO 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

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

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. 55) for example code.

Related PRP Actions:

Send Peripheral

PMDPeriphWrite

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
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

PMDputch

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. 59) 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

PMDputs

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. 59) 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. 29)), or dual-ported RAM devices (see **PMDMemoryOpen32** (p. 42)), peripherals on the device (see **PMDPeriphOpenCOM** (p. 48), **PMDPeriphOpenTCP** (p. 53), **PMDPeriphOpenUDP** (p. 53), **PMDPeriphOpenISA** (p. 49), and **PMDPeriphOpenCAN** (p. 46)) .

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

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 **PMDTaskWaitUntil** (p. 63).

Related PRP Actions:

none

PMDTaskWaitUntil

C-Motion Engine | Host-Based

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

PMDWaitForEvent

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 must be a Magellan attached device.

If an asynchronous event notification is received for any of the Magellan axes of the motion processor attached to the device then the function returns and the axis and event status register are 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 **PMD_ERR_TIMEOUT** is returned, and **hEvent** is not written. A **timeout** value of **PMD_WAITFOREVER** (ffff) disables the time out.

Asynchronous event notification is an optional Magellan feature described in the *Magellan Motion Control IC User Guide*. The conditions causing an event notification are programmable, using commands described in the *C-Motion Magellan Programming Reference*. The **PMDWaitForEvent** function handles all the necessary function calls to deal with the event except for the **PMDClearInterrupt** function. 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:

none

This page intentionally left blank

4. PRP Action Reference

In This Chapter

- Action Table - Code Order
- Action Table - Alphabetical Order

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.

4.1 Action Table - Code Order

Action	Resource	Sub-action	C Procedure
NOP	*any*		
Reset	Device		PMDDeviceReset
	MotionProcessor	PMDDeviceReset	
Command	CMotionEngine	Flash	
		Task	PMDTaskStart
			PMDTaskStop
	MotionProcessor		*Any C-Motion Commands*
Open	Device	MotionProcessor	PMDAxisOpen
		CMotionEngine	PMDRPDeviceOpen
		Memory32	PMDMemoryOpen32
		PIO	PMDPeriphOpenPIO
		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	PMDGetTaskState
	Device	Default	PMDGetDefault
		ResetCause	PMDMBGetResetCause
		Version	PMDDeviceGetVersion

4.2 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	PMDTaskStart PMDTaskStop
	MotionProcessor		*Any C-Motion Commands*
Get	CMotionEngine	Console	
		TaskState	PMDGetTaskState
	Device	Default	PMDDeviceGetDefault
		ResetCause	PMDMBGetResetCause
		Version	PMDDeviceGetVersion
NOP	any		
Open	Device	CAN	PMDPeriphOpenCAN
		CMotionEngine	PMDRPDeviceOpen
		ISA	PMDPeriphOpenISA
		Memory32	PMDMemoryOpen32
		MotionProcessor	PMDAxisOpen
		COM	PMDPeriphOpenCOM
		PIO	PMDPeriphOpenPIO
		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

This page intentionally left blank.

Close *various*

Coding:

action	sub-action	resource
4	-	various

Arguments: none

Return Data: none

Packet Structure:

<table>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">4</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

<table>
<tr><td>write</td><td colspan="3">resource</td><td colspan="5">address</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

<table>
<tr><td>read</td><td colspan="2">2</td><td colspan="2">status</td><td colspan="4"><i>reserved</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

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 *hPeriph);
PMDresult PMDDeviceClose(PMDDevice *hDevice);
PMDresult PMDMemoryClose(PMDMemory *hMemory);
```

CommandFlash CMotionEngine

Coding:

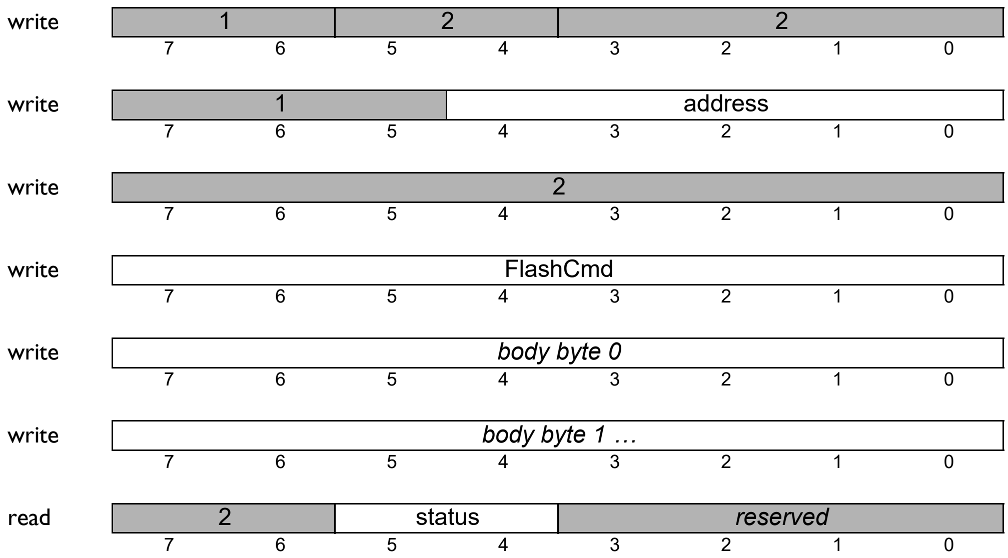
action	sub-action	resource
2	2	1

Arguments:

name	instance	encoding
FlashCmd	FlashStart	1
	FlashData	2
	FlashEnd	3

Returned Data: none

Packet Structure:



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.

Command Task CMotionEngine

Coding:

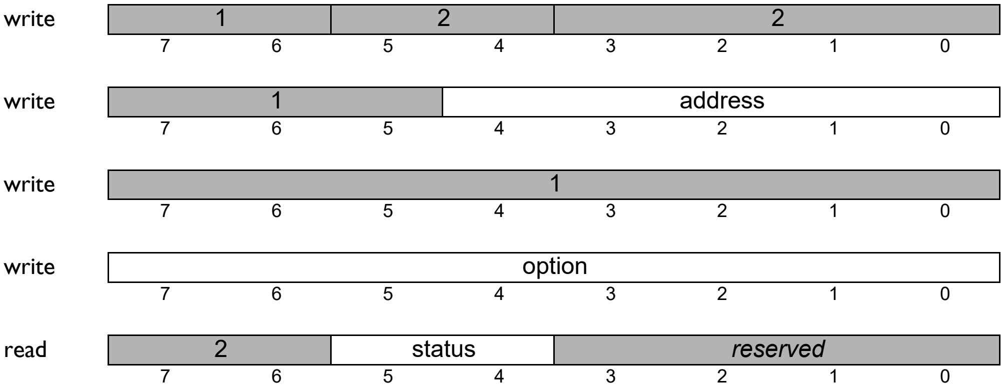
action	sub-action	resource
2	1	1

Arguments:

name	instance	encoding
option	1	start
	2	stop

Returned Data: none

Packet Structure:



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 PMDTaskStart(PMDDeviceHandle *pDevice);
PMDresult PMDTaskStop(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	*Magellan command code*							
write	rxCount		*Magellan axis*					
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 *C-Motion Magellan Programming Reference,* 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 the command word is an opcode for the Magellan command. The second byte 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. 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 *C-Motion Magellan Programming Reference,* 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.

GetConsole 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.

Get Default Device

Coding:

action	sub-action	resource
10	2	0

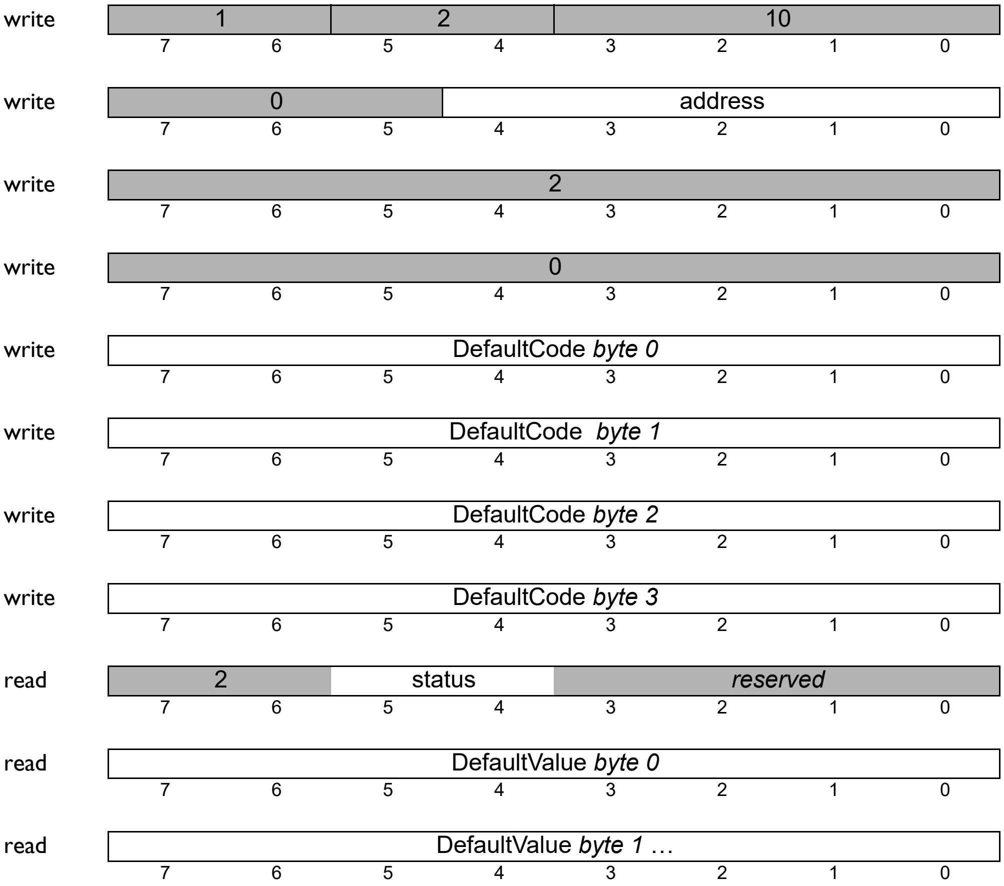
Arguments:

name	type	meaning
DefaultCode	unsigned 32 bit	default identifier

Returned Data:

name	type	meaning
DefaultValue	varies	varies – see Set ValueDefault

Packet Structure:



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 109 for a ta‘ble 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:

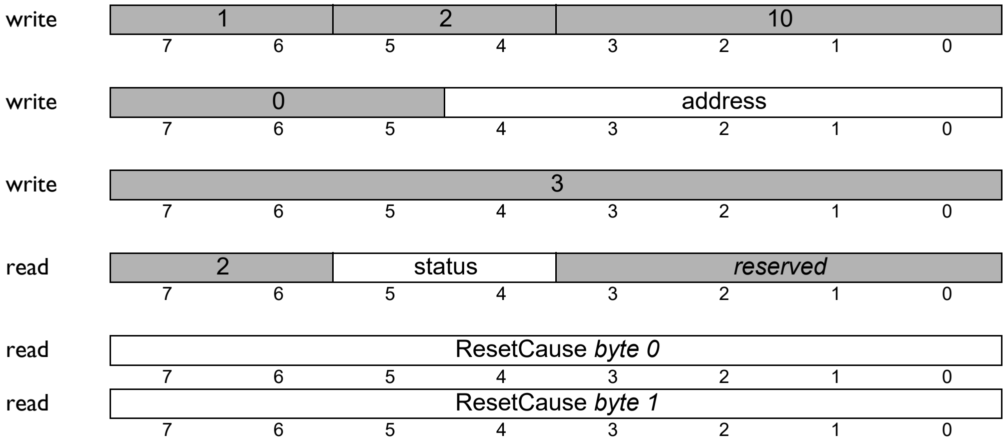
action	sub-action	resource
10	3	0

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:



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 *C-Motion Magellan Programming Reference* for procedure documentation.

GetTaskState 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 PMDGetTaskState(PMDDeviceHandle *pDevice,
PMDuint32 *state);
```

GetVersion Device

Coding:

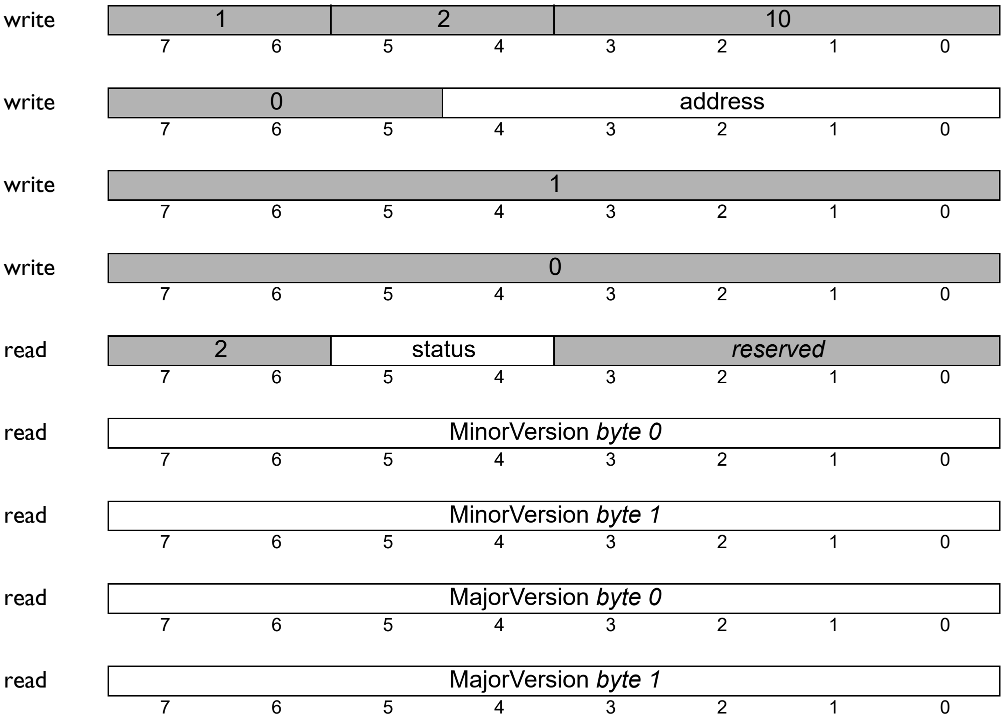
action	sub-action	resource
10	1	0

Arguments: none

Returned Data:

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

Packet Structure:



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

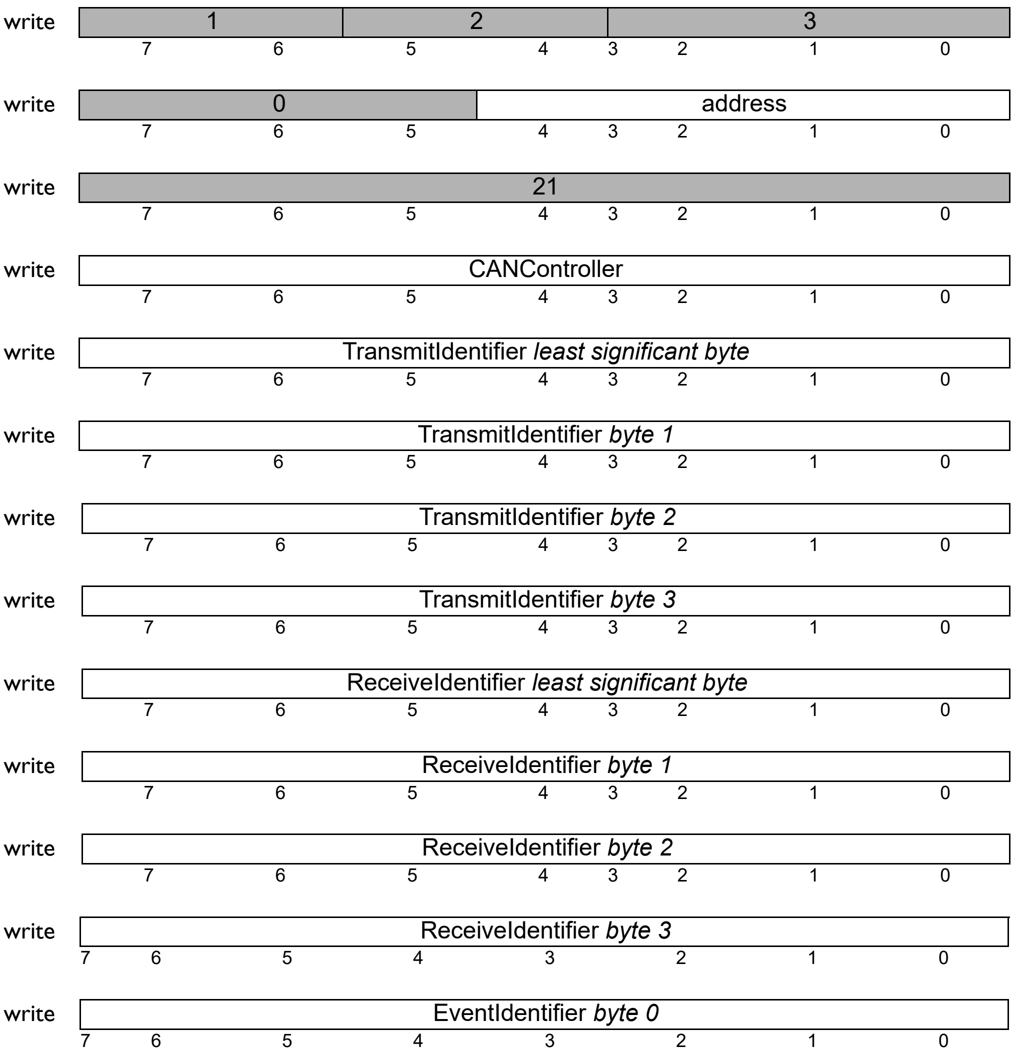
Arguments:

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	*Reserved*							
	7	6	5	4	3	2	1	0

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

write	*Reserved*							
	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 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.

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:

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

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,
                          PMDDataSize datasize,
                          PMDMemoryAddress memoryaddress);
```

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:

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.

Open COM 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:

<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">3</td></tr>
<tr><td>write</td><td colspan="3">0</td><td colspan="5">address</td></tr>
<tr><td>write</td><td colspan="8">20</td></tr>
<tr><td>write</td><td colspan="8">SerialPort</td></tr>
<tr><td>write</td><td colspan="5">multidrop address</td><td colspan="2">0</td><td>protocol</td></tr>
<tr><td>write</td><td>protocol</td><td>stop bits</td><td colspan="2">parity</td><td colspan="4">transmission rate</td></tr>
<tr><td>read</td><td colspan="2">2</td><td colspan="2">status</td><td colspan="4"><i>reserved</i></td></tr>
<tr><td>read</td><td colspan="8">PeriphAddress</td></tr>
</table>

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);
```

OpenPIO 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 PIO 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 PMDPeriphOpenPIO(
                           PMDPeriphHandle*hPeriph
                           PMDDeviceHandle*hDevice,
                           WORD address,
                           BYTE EventIRQ,
                           PMDDataSize datasize);
```

OpenTCP Device

Coding:

action	sub-action	resource
3	22	0

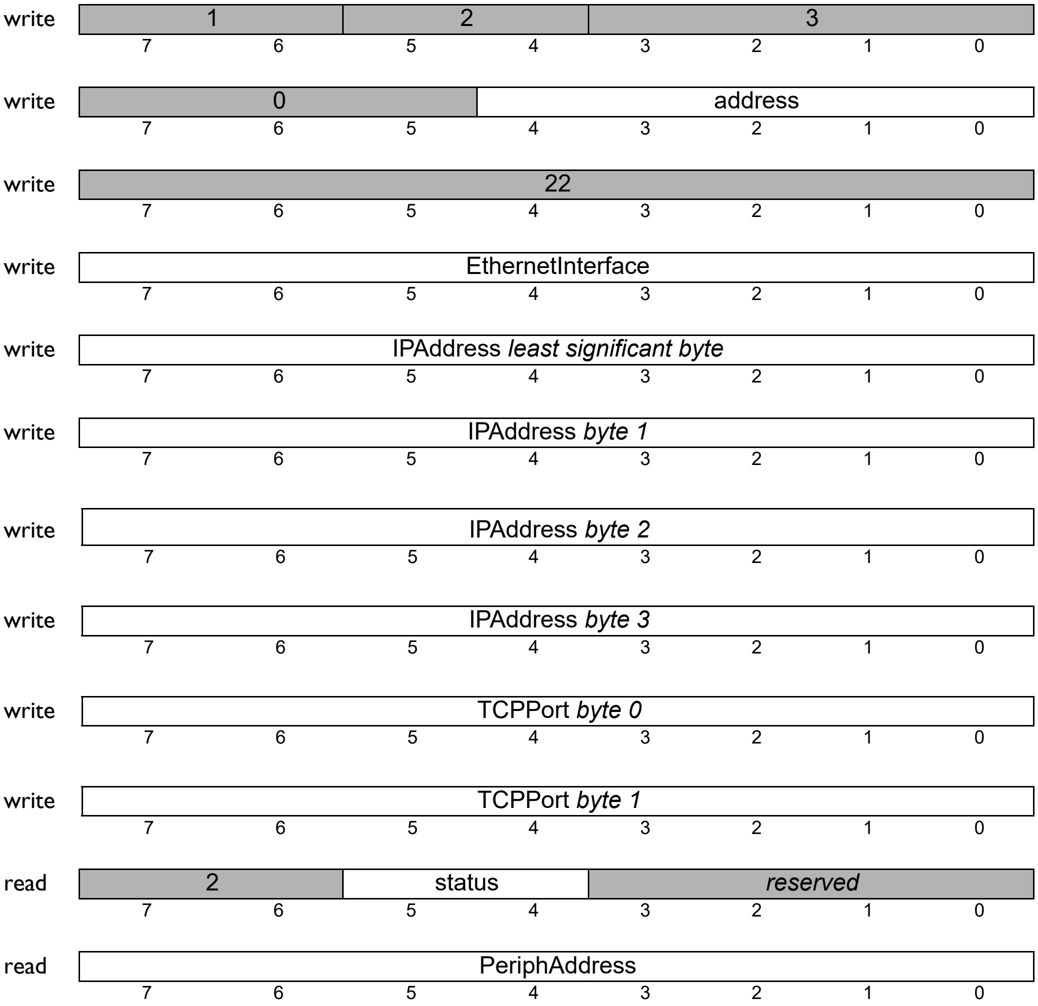
Arguments:

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:



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

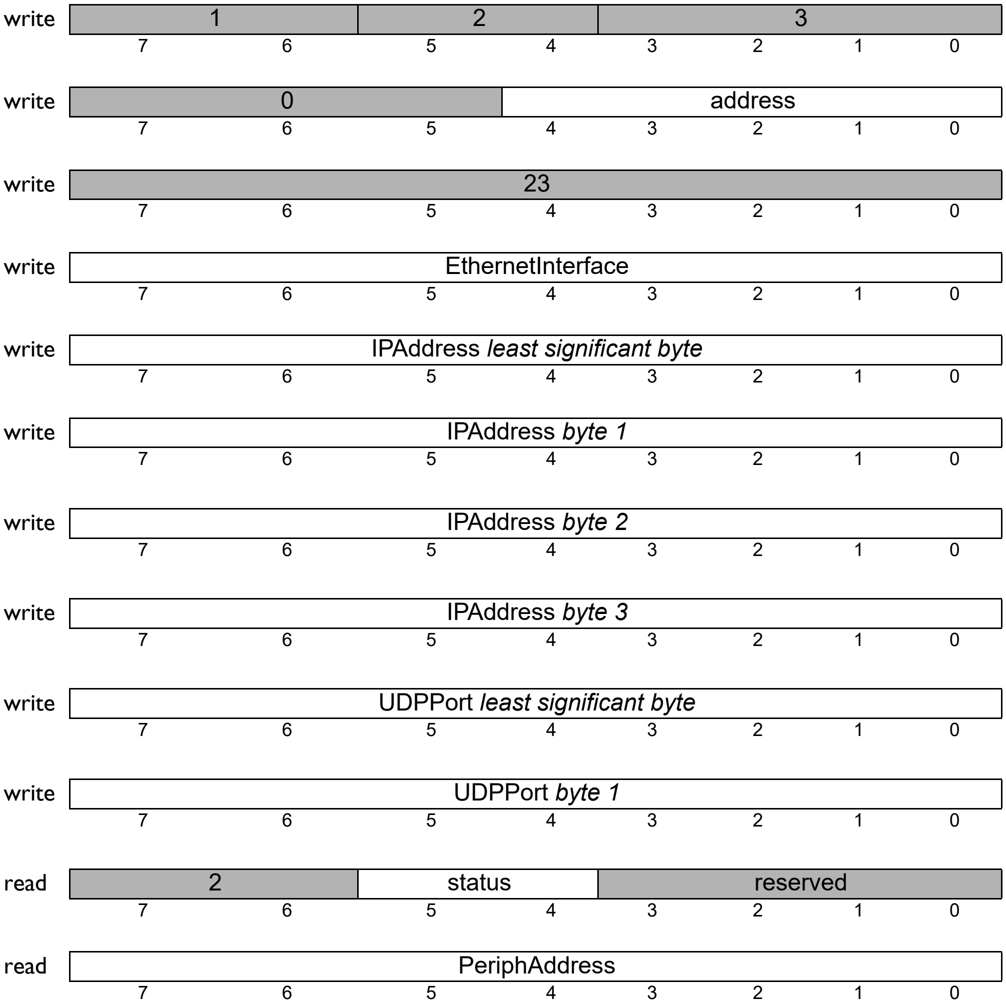
Arguments:

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:



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);
```

OpenMotionProcessor Peripheral

Coding:

action	sub-action	resource
3	0	4

Arguments: none

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	0							
write	0							
read	2		status		*reserved*			
read	RemoteAddress							

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);
```

Open MultiDrop 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:

<table>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">3</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="3">4</td><td colspan="5">address</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="8">25</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="8">MultiDropAddress</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>read</td><td colspan="2">2</td><td colspan="2">status</td><td colspan="4"><i>reserved</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>read</td><td colspan="8">RemoteAddress</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

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 Control IC User Guide.*

C language syntax:

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

Read Byte Peripheral

Coding:

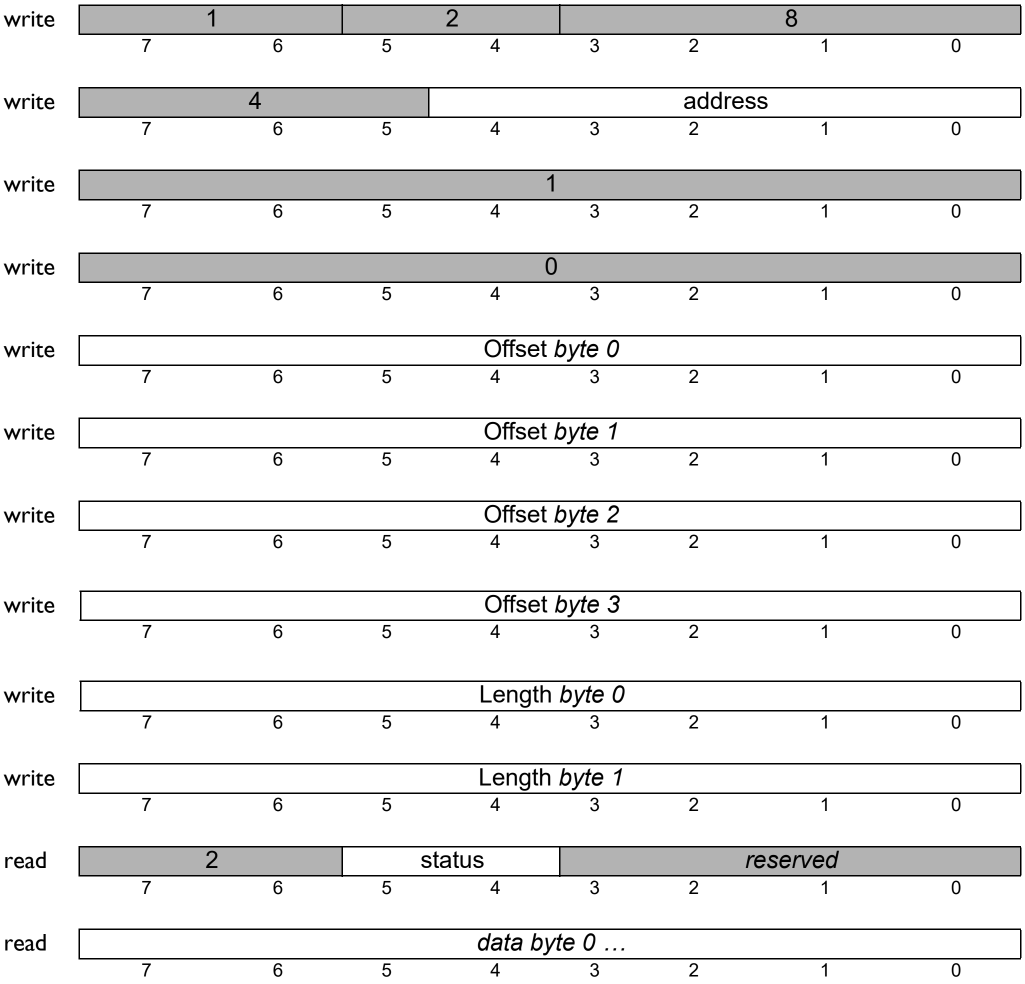
action	sub-action	resource
8	1	4

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:



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:

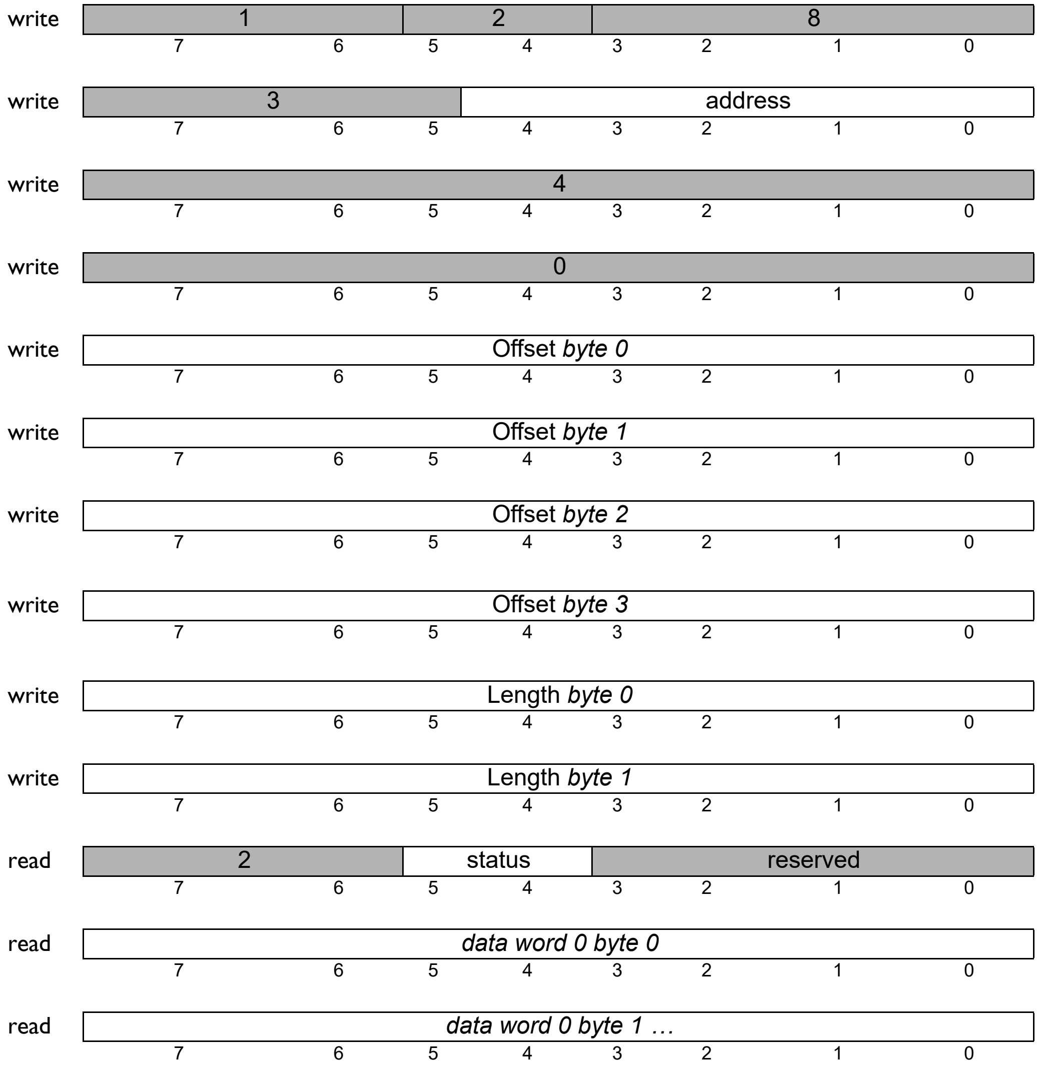
action	sub-action	resource
8	4	3

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:



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 (PMDMemoryHandle *hMemory,
                         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:

<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">8</td></tr>
<tr><td>write</td><td colspan="3">4</td><td colspan="5">address</td></tr>
<tr><td>write</td><td colspan="8">2</td></tr>
<tr><td>write</td><td colspan="8">0</td></tr>
<tr><td>write</td><td colspan="8">Offset <i>byte 0</i></td></tr>
<tr><td>write</td><td colspan="8">Offset <i>byte 1</i></td></tr>
<tr><td>write</td><td colspan="8">Offset <i>byte 2</i></td></tr>
<tr><td>write</td><td colspan="8">Offset <i>byte 3</i></td></tr>
<tr><td>write</td><td colspan="8">Length <i>byte 0</i></td></tr>
<tr><td>write</td><td colspan="8">Length <i>byte 1</i></td></tr>
<tr><td>read</td><td colspan="2">2</td><td colspan="2">status</td><td colspan="4"><i>reserved</i></td></tr>
<tr><td>read</td><td colspan="8"><i>data byte 0 …</i></td></tr>
</table>

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);
```

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:

<table>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">6</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="3">1</td><td colspan="5">address</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="8">timeout <i>byte 0</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>write</td><td colspan="8">timeout <i>byte 1</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>read</td><td colspan="2">2</td><td colspan="2">status</td><td colspan="4">reserved</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>read</td><td colspan="8"><i>received data byte 0</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>read</td><td colspan="8"><i>received data byte 1 …</i></td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

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. 107) 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:

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

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.

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:

<table>
<tr><td>write</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="4">1</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

<table>
<tr><td>write</td><td colspan="3">0</td><td colspan="5">address</td></tr>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

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:

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

write	2			address				
	7	6	5	4	3	2	1	0

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

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:

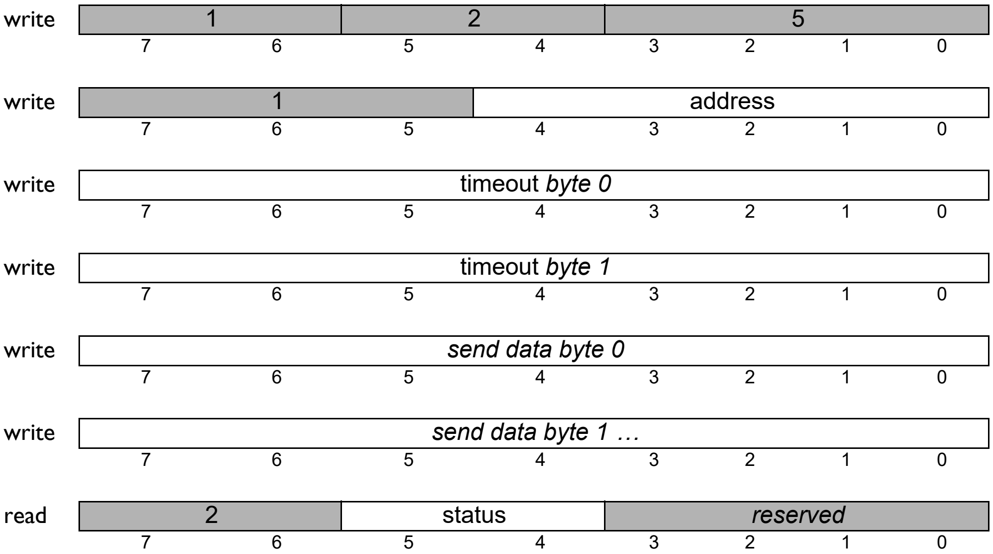
action	sub-action	resource
5	-	1

Arguments:

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

Returned Data: none

Packet Structure:



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:

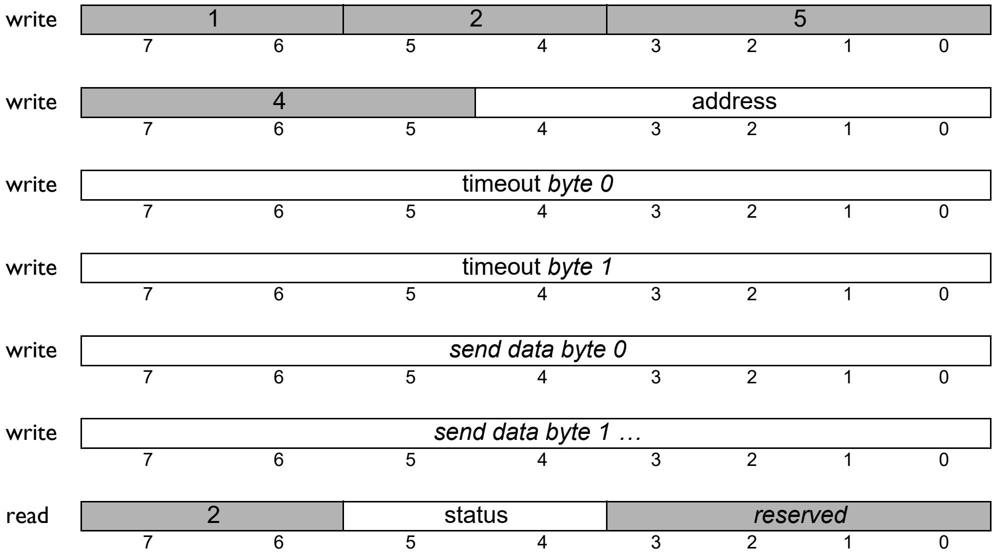
action	sub-action	resource
5	-	4

Arguments:

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

Returned Data: none

Packet Structure:



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:

```
PMDResult PMDDeviceSetConsole(PMDDeviceHandle *hDevice,
                              PMDPeriphHandle *hPeriph);
```

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:

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

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
8		*reserved*	
9	PRP		
>9		*reserved*	

C language syntax:

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

Write DWord Memory

Coding:

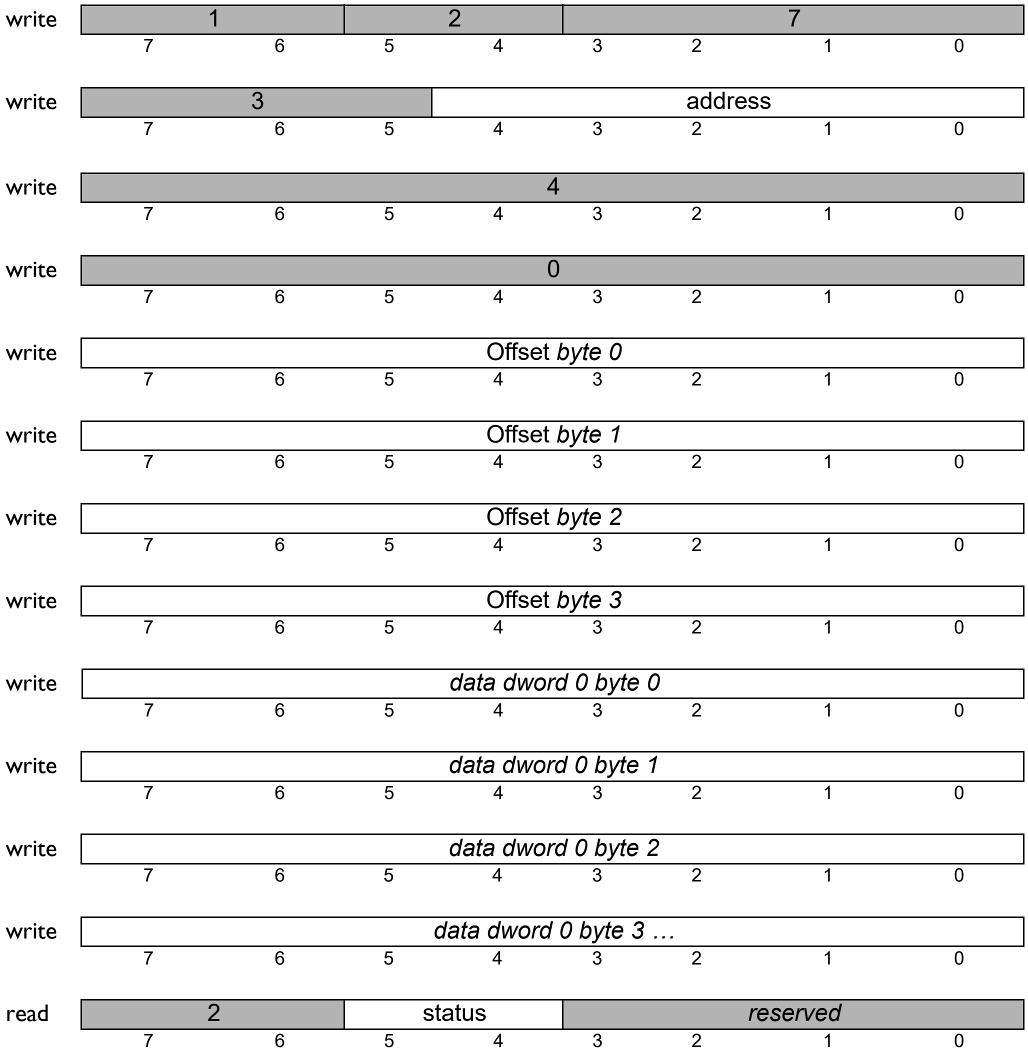
action	sub-action	resource
7	4	3

Arguments:

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

Returned Data: none

Packet Structure:



Description:

The **Write DWord Memory** 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:

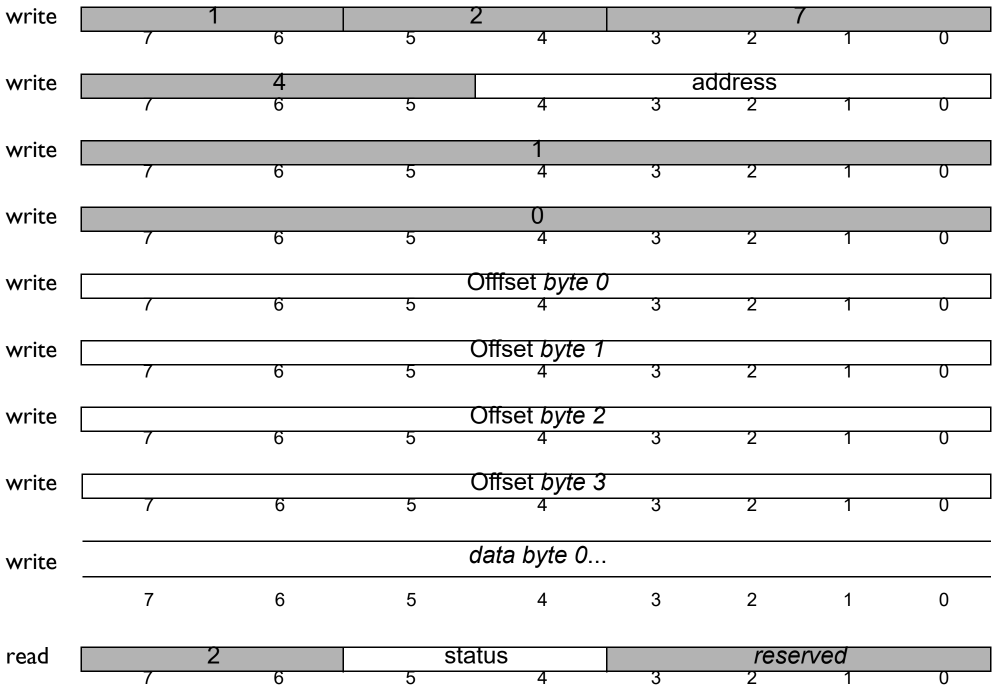
action	sub-action	resource
7	1	4

Arguments:

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

Returned Data: none

Packet Structure:



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);
```

Appendix A. PRP Transport

In This Appendix

- PRP Transport Over Serial
- PRP Transport Over TCP/IP
- PRP Transport Over CAN

PRP may be transported using a serial, TCP/IP, CAN, or SPI communication channel. This section discusses these communication channel-specific aspects of PRP message transport and processing.

A.1 PRP Transport Over Serial

To transport PRP packets over serial a header is used to specify the length of the PRP packet and to detect most cases of packet corruption.

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 packet formats for the two cases:

Point-to-Point Serial Packet

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

Multi-Drop Serial Packet

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

7	6	5	4	3	2	1	0
PRP packet *byte l* …							
7	6	5	4	3	2	1	0

The MultiDropAddress field is used to address a particular 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 bytes. For example if there are 2 PRP packet bytes to be transported the length field value is 2.

The checksum field is a simple additive checksum modulo 256, over just the bytes in the PRP packet. For example if there are 2 PRP packet bytes to be transported then the checksum is calculated over these 2 bytes.

Both outgoing and response packets are formatted in the same way.

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, 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 a PRP error response packet with an error code of **PMD_ERR_RP_Checksum**. See Section 2.5.2, PRP Response Packet for a list of PRP response packet error codes.

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.

A.2 PRP Transport Over TCP/IP

PRP packets are realized as TCP/IP packets. Three padding bytes are added to the beginning of the response packet and can be ignored. For example if the PRP response packet is two bytes in length, the 1st, 2nd, and 3rd bytes of the TCP/IP response packet would hold zero, and the 4th and 5th bytes would hold the PRP response packet.

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**.

A.3 PRP Transport Over CAN

PRP over CAN 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.

CAN messages are limited to eight bytes of data, which means that some PRP packets may require several CAN messages for complete transport. In order to support this a segment/de-segment protocol is used. 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.

A header byte added as the first byte of each CAN message is used for segment identification. All of the remaining (up to 7) bytes are used for the PRP packet content. Each CAN message used for PRP is either an *initial* message, or a *continued* message. An initial message is the first message and is followed by zero or more continued messages, which complete the PRP packet content.

The header byte of the 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, and each subsequent 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 will immediately send a PRP error packet with the error code **PMD_ERR_RP_InvalidPacket**. Each continued message must be sent within 100ms otherwise the PRP packet processing state machine will be reset.

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 including the segment header bytes. Reception of the expected sequence numbers is very good evidence that a packet has been correctly received.

Example

To send the 17 byte PRP packet 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 the message-by-message CAN content is:

1st CAN message (all values in hex):

82, 01, 02, 03, 04, 05, 06, 07

2nd CAN message:

01, 08, 09, 0A, 0B, 0C, 0D, 0E

3rd CAN message:

02, 0F, 10, 11

This page intentionally left blank.

Index

This page intentionally left blank.

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

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

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

P M D

Performance Motion Devices
80 Central Street
Boxborough, MA 01719