Reflective Memory Driver & System Service

Revision: 1.3

Date: 23-JUL-2008

Technical Writer:
Earl D. Lakia
Senior Staff Engineer
IPACT Inc.

file: W:\Projects\Active\IPACT\GEFANUC_RFM\Documentation\RFM_Manual.doc
date: 7/23/2008 7:41 AM

Document Revision History	
Date	**Description**
25-Sep-2006	E. Lakia Orginal writing
27-Sep-2006	M. Taylor Added header and page numbers & did some editing.
28 & 29-Sep-2006	M. Taylor Did some editing.
01-Nov-2006	E. Lakia Update/cleanup
02-Nov-2006	E. Lakia Added RFM_PEEK and RFM_POKE utilities
14-Nov-2006	E. Lakia Added DEVDMP utility and some updates regarding the logical names for different modes of operation.
19 - 24-Apr-2007	M. Taylor Did some editing.
1-May-2007	M. Taylor Inserted index
2-May-2007	E. Lakia Added PSTCFG documentation
2-May-2007	M. Taylor Updated drawings
18-Sep-2007	E. Lakia Adds some post installation information, including the license information.
07-Nov-2007	M. Taylor Inserted a drawing.
14-dec-2007	E. Lakia Document cleanup
02-mar-2008	E. Lakia Documentation correction for RFM_OPEN call. Added the RFM_Read and RFM_Write function calls.
23-jul-2008	E. Lakia Model number change by GEFANUC (e.g., dropping the legacy VMIC from the model)

1 Introduction

This document explains the operation and functionality of the GEFANUC PCI-5565 PCI reflective memory interface and associated OpenVMS system services. The user should reference section: 2 for the installation procedure. The following is a brief description of the environment.

- OpenVMS install kit (utilizes sys$udpate:vmsinstal.com)
- OpenVMS PCI device driver- Used to control the actual GEFANUC PCI 5565 device and configure it into OpenVMS operating system running on HP Integrity or Alpha servers.
- System Service- The system service is a privileged image that provides a controlled access to the reflective memory contained within the PCI 5565 and the reflective memory ring using the above mentioned device driver. Access to the reflective memory can be either address based or symbolic tag based access for applications running on the OpenVMS host. The system service also provides the ability to run in three modes (e.g., normal, read only, and ghost).
- Configuration Application- This application provides the mapping of the tags to memory addresses. The configuration uses a comma delimited file to provide a symbolic mapping of process variables within the memory. Typically this is exported from an Excel or spreadsheet database.
- Debug/Development Utilities- A set of utilities is provided that allow for the debug and development of systems based using reflective memory.

2 Installation

The reflective memory installation uses the OpenVMS “VMSINSTAL” method for installation. The system supports up to four PCI cards within the PCI or PCIX bus. The PCI-5565 reflective memory ring adapter follows the PCI standard, which allow for easy configuration into the system. Each of the PCI-5565s should be configured for a unique node IDS for each reflective memory ring and installed into the PCI back-plane of the Integrity or Alpha system prior to software installation. The PCI or PCIX bus is probed by as part of the power up and self-test code of the Integrity or Alpha Hewlett Packard Servers. This configuration data is stored in the adapter tables and read as part of the kit installation. The kit can be installed without the card installed in the bus, but custom startup editing will be required after the install.

The OpenVMS install kit is provided as a saveset on a CD or downloaded via FTP that must be copied to sys$update prior to installation and the file attributes set (done by the MOVE_KIT.COM command procedure). The following is a typical procedure where DKA400 is the CD drive on the OpenVMS system and the user is logged into the system account.

```
$ SET DEF SYS$UPDATE:
$ Mount DKA400:/over=id
$ @DKA400:MOVE_KIT.COM
```

The user should invoke VMSINSTAL.COM and provide the product kit, which was copied, from the CD (also printed by MOVE_KIT.COM). The following is a session capture of the kit installation. A single PCI-5565 PCI card was present in the PCIX bus. After installation, a single command procedure is added to target directory that can be used to load the device driver and the system service (below if for demonstration release 1.1):

```
SYS$COMMON:[IPD5565D011]RFM_5565_STARTUP.COM
```

This is normally added to the system’s sys$startup:systartup_vms.com command procedure. The utility RFM is normally ran at this time to set the writeable regions for the Alpha or Itanium host for each reflective memory adapter.

The name of the kit reflects if the kit is a demonstration and evaluation kit or a production kit:

IPD5565PRRV.A = production
IPD5565DRRV.A= demonstration

The “RRV” are the standard OpenVMS release and version number conventions and the “.A” is the first saveset of the kit. The production versions require a serial number and license number from IPACT.

The kit is not supplied with an IVP (Installation Validation Procedure) as doing so might risk the corruption of the memory on the reflective memory by invalid writes. However, when the driver is loaded, informational messages are written to the operator console and the operator log file (sys$manager:operator.log, see section: 4.17). The utility DEVDMP can also be used to check if the driver was successfully loaded (this utility requires change mode to Kernel privilege).

2.1 VMS INSTAL Session Capture

The following is capture of an actual installation for a demonstration kit. The user would have been prompted for a serial number and license number for a production version.

```
$ set def sys$update
$ @vmsinstal


        OpenVMS  Software Product Installation Procedure V8.2-1


It is 25-SEP-2006 at 08:57.

Enter a question mark (?) at any time for help.

* Are you satisfied with the backup of your system disk [YES]? y
* Where will the distribution volumes be mounted: sys$update:

Enter the products to be processed from the first distribution volume set.
* Products: IPD5565D011
* Enter installation options you wish to use (none):

The following products will be processed:

  IPD5565D V1.1


        Beginning installation of IPD5565D V1.1 at 08:58

%VMSINSTAL-I-RESTORE, Restoring product save set A ...

IPACT Process Database for GE FANUC VMIC PCI5565 Adapter

Copyright
      IPACT Inc.
      260 S. Campbell St.
      Valparaiso, IN  46485

      www.ipact.com

      Support: 219-464-7212
               mpblus@ipact.com


This software is licensed to a single computer.

This is a demo kit.  The driver will only function for
```

```
two days after the driver is loaded.
%IPD-I-KitType This is a Demonstration kit

This software will be installed in the following
directory:
  sys$common:[IPD5565D011]
  sys$common:[IPD5565D011.EXAMPLES]


All user input complete

%IPD-S-CoffeeBreak All input complete
%IPD-I-Linking 5565 device driver

%IPD-I-Linking RFMSS system service

%IPD-I-Linking DEVDMP utility

%IPD-I-Linking DUMPMCF utility
%IPD-I-Linking FINDPCI utility

%IPD-I-Linking GETIT utility
%IPD-I-Linking PSTCFG utility
%IPD-I-Linking PUTIT utility
%IPD-I-Linking RFM

Looking for 5565 PCI adapter cards
Expect output showing system bus structure
Running FINDPCI to find your devices

%IPD-I-Find5565 Find PCI adapters


Changing mode to Kernel to find PCI RFM devices
 Reflective Memory PCI device found, 1

      Vendor id:   0x114a
  PCI Device ID:   0x5565
   SubVendor ID:   0x10b5
   SubSystem ID:   0x9656
     Controller:   0x0
            CSR:   0x0
    Node Number:   0x6008
         Vector:   0x1530
            IRQ:   0x1530
            CRB:   0x0
            ADP:   0x0

               PCI Devices found
Entry     Adp       Ba     Vendor  DevId   SubVendId  SubSysId
 01) 0x89450100 0x89485728 1011   0005      0000      0000
 02) 0x89450100 0x89485838 1095   0649      1095      0649
 03) 0x89450100 0x89485948 8086   1229      103c      1274
 04) 0x8946e740 0x89487a58 1033   0035      1033      0035
 05) 0x8946e740 0x89487b68 1033   0035      1033      0035
 06) 0x8946e740 0x89487c78 1033   00e0      1033      00e0
 07) 0x8946b7c0 0x89489b28 1011   0005      0000      0000
 08) 0x8946b7c0 0x89489c38 14e4   1645      103c      12a4
 09) 0x89450600 0x8948bc58 1000   0030      1000      1000
 10) 0x89450600 0x8948bd68 1000   0030      1000      1000
 11) 0x89464000 0x89493e28 114a   5565      10b5      9656
 12) 0x89492400 0x89497aa8 1014   01a7      0000      0000

Creating new Driver install command file
  vmi$kwd:rfm_driver_load.com
 RFM board: 1
$ MCR SYSMAN IO CONNECT RBA0: -
    /DRIVER_NAME=SYS$LOADABLE_IMAGES:SYS$RFMDRIVER -
    /CSR=0
    /NODE=%x6008
    /ADAPTER=9 (TR number)
    /VECTOR=%x1530
$ set prot=(s:rwpl,o:rwpl,g:rwpl,w:rwpl)/device rmA
%IPD-I-Moving files

%VMSINSTAL-I-SYSDIR, This product creates system directory [IPD5565D011].
%CREATE-I-EXISTS, VMI$COMMON:[IPD5565D011] already exists
%VMSINSTAL-I-SYSDIR, This product creates system directory [IPD5565D011.EXAMPLES
].
%CREATE-I-EXISTS, VMI$COMMON:[IPD5565D011.EXAMPLES] already exists

IPD-S-Installed Product successfully installed
%VMSINSTAL-I-MOVEFILES, Files will now be moved to their target directories...

        Installation of IPD5565D V1.1 completed at 08:58

    Adding history entry in VMI$ROOT:[SYSUPD]VMSINSTAL.HISTORY

    Creating installation data file: VMI$ROOT:[SYSUPD]IPD5565D011.VMI_DATA

Enter the products to be processed from the next distribution volume set.
* Products:

        VMSINSTAL procedure done at 08:59
```

2.2 POST Install

After installing the kit, the command procedure to load the driver and install the system service should be ran:

```
$ @SYS$COMMON:[IPD5565D011]RFM_5565_STARTUP.COM
```

If the user intends to use the utilities, the foreign commands for the utilities are defined in the command file:

```
$ @IPD__PROD:RFM_SYMBOLS.COM
```

All files for this product are located in the installation directory including the device driver and its symbol table.

If the 5565 driver had not been installed in the PCIX bus when the kit was installed, the user may look for the driver again by executing the following from within the IPD__RFM: directory.

```
$ Set Def IPD__RFM:
$ FINDPCI  IPD__RFM:RFM_DRIVER_LOAD.COM
```

OpenVMS does not support the reloading of a device driver. If a new release of the driver is installed, the system must be bootstrapped.

3 IPD System Service

The actual writing and reading of the reflective memory ring data requires PCI bus memory cycles. This is a privileged function within the OpenVMS executive as errant read and writes can cause system faults. OpenVMS provides a controlled method to accomplish this using a device driver. The system service is installed with privileges that allow the normal user the ability to write to the reflective memory. The system service checks to ensure that all writes to the reflective memory are within the writeable region for this node by each adapter (e.g., PCI-5565). The system service supports development when a PCI-5565 is not available and when a ring member is absent. The system service calls the PCI_5565 device driver to actually access the reflective memory (unless in one of the offline modes). If partitions have been enabled in the driver, the validation of the writeable area is done by the driver.

The API is designed to support two access methods:

- Read and write directly to memory locations in the reflective memory (e.g., Peek and Poke).
- Read and write symbolic tags to memory locations in reflective memory.
- Provides method for notification of events on the reflective memory ring

The use of the symbolic tag reference requires the configuration of each symbolic tag using a simple comma delimited file (CSV) typically created using a spreadsheet (example spread sheet is provided on the distribution CD). This method allows for the support of a simple data type such as an integer as well as a more complex type such as a structure. Using this method allows support for some of the utilities (GETIT and PUTIT) and debug methods described in the utilities section. All memory data types should be long-word aligned where possible, including those data types, which are shorter (e.g., pad to a four byte modulus).

The system service is linked into a user application with the following linker option command: "ipd__prod:rfmss.exe/share". The logical names ipd__prod and rfmss are defined by the RFM_5565_STARTUP.COM command procedure. A typical linker command might be the following (example from rfm_poke, ipd_library.olb contains utility routines used by rfm_poke, all objects for the utility are in rfm_poke.olb). The "ipd__prod:ipd_library.olb/include=ipdmsgdefinition" includes message codes for the RFMSS system service.

```
$link/exe=rfm_poke,sys$input/opt
rfm_poke.olb/include=rfm_poke
rfm_poke.olb/lib
ipd__prod:ipd__library.olb/lib
ipd__prod:ipd__library.olb/include=ipdmsgdefinition
ipd__prod:rfmss.exe/share
```

The following lists each of the API (Application Programming Interface) calls provided in the system service. These are individually documented later in this chapter. Typically a user does one of the following (also see provided example applications provided in the distribution examples directory):

1. Calls the RFM_OPEN to connect to the driver, and calls the RFM_READ and RFM_WRITE as needed

2. Calls the RFM_OPEN to connect to the driver, calls RFM_REFERENCE_TAG to get information about the tags that are going to be used, and calls the RFM_READTAG and RFM_WRITETAG as needed

3. Makes call directly to the PCI-5565 device driver using the QIO interface.

RFM_OPEN- Open a particular PCI-5565 reflective memory adapter

RFM_READ- Read data from reflective memory at supplied offset

RFM_WRITE- Write data to reflective memory at supplied offset

RFM_READTAG- Read data for a tag, location of tag acquired from RFM_REFERENCE_TAG.

RFM_WRITETAG- Write data to a tag, location of tag acquired from RFM_REFERENCE_TAG.

RFM_REFERENCE_TAG- Reads configuration file for a tag to acquire its data type, length, and location in reflective memory.

RFM_EXTENDED_STATUS- Returns additional information when RFM_READTAG and RFM_WRITETAG are used.

RFM_WRITE_BLIND- Write to a TAG without regard to node’s writeable region (used for simulation of an absent ring member).

The diagram below shows the basic implementation of the system service.

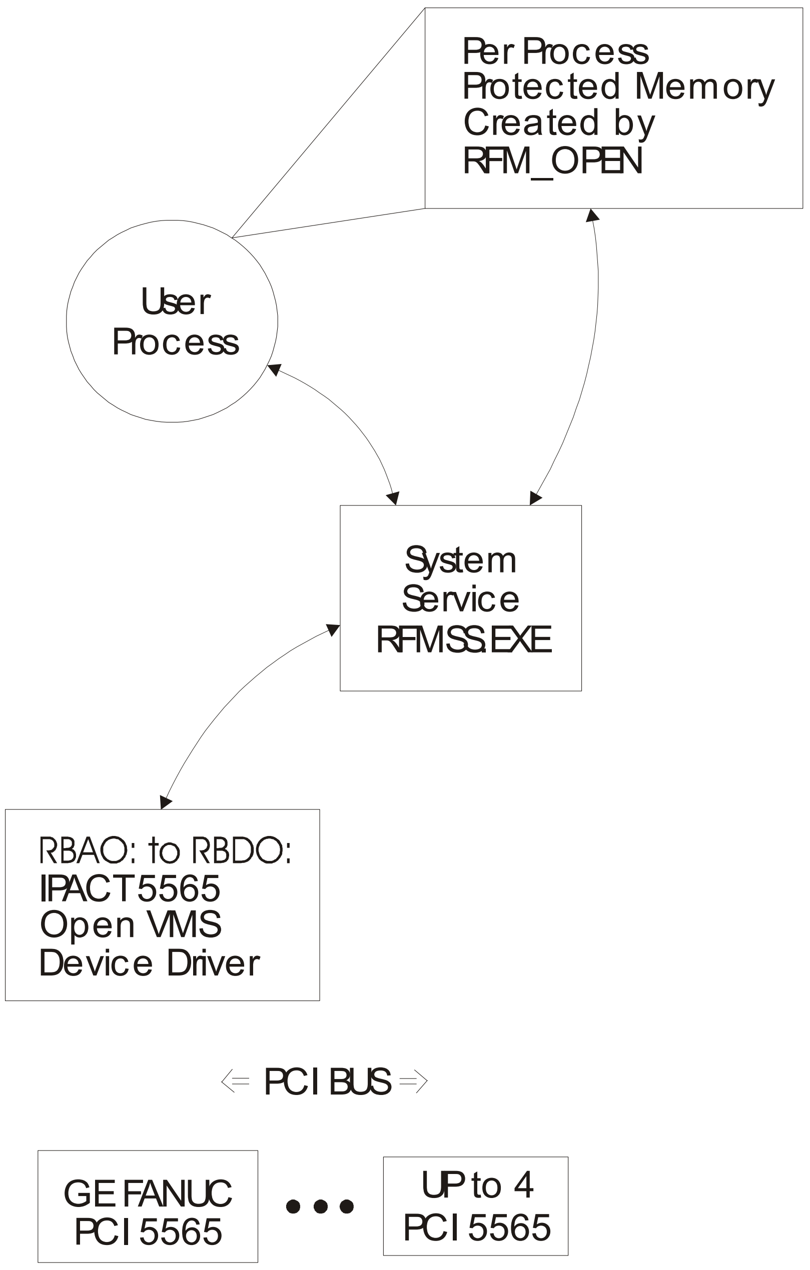


Figure 3-1 IPD System Services

3.1 VMIC INTERFACE OPERATING MODES

There are three modes of operation provided by the IPDRFMSS system service layer. The modes are referred to as “NORMAL”, “READONLY”, and “OFFLINE” modes. These modes are implemented in such a way that no change is required to the applications using the IPDRFMSS system service function. The mode of operation is controlled by a single GROUP logical name called IPDRFM_MODE. Assigning the value of NORMAL, READONLY, or OFFLINE to this logical will cause applications to operate in the defined mode. The desired mode of operation must be defined prior to starting the VMIC "aware" applications on the OpenVMS system. If a mode change is required, each VMIC "aware" application on the OpenVMS system must be restarted following the mode change.

The range of VMIC Reflective memory address space that can be written by the OpenVMS system service is written by the RFM utility. The device driver stores this information and validates writes prior to doing any writes to the Reflective memory.

In the NORMAL mode of operation, all applications which use the IPDRFMSS functions, will write to and read from the specified tags on the VMIC ring. This is the standard operating mode.

The READONLY mode is provided to allow applications to read actual live VMIC data from the ring, while blocking writes to the ring from the OpenVMS node. The write operations place the output data into a the global memory section. This mode is ideal for testing the response of applications to external signals.

The OFFLINE mode is provided to allow total isolation from the VMIC ring. In this mode, read and write operations are from a global memory section.

The following three figures show the operation of the interface routines in each of the three valid operating modes. The READONLY and OFFLINE modes require the following system logical names:

Mode logical name is the adapter device name plus "MODE". The value of this logical name should be NORMAL, OFFLINE, or READONLY. If this logical name is not defined, then NORMAL is assumed.

Region name logical is the name of the region to be used for READONLY and OFFLINE usage. It is defined as the ADAPTER plus GHOST_REG (e.g., RBA0GHOST_REG)

Section filename logical is the RMS backing page file for the OFFLINE global section. It is defined as the ADAPTER plus GHOST_SECFILE (e.g., RBA0_GHOST_SECFILE)

Ghost Region Size logical is the size of the OFFLINE global section. It is defined as the ADAPTER plus GHOST_SIZE (e.g., RBA)_GHOST_SIZE).

To set up a global GHOST region for a 64 MB reflective memory card the following would be required:

```
$ create ipd__prod:64M_PAGE.FDL ipd__prod:64m.pag
$ set file/end_of_file ipd__prod:64m.pag
$ define/system RBA0GHOST_SECFILE “ipd__prod:64m.pag”
$ define/system RBA0GHOST_REG RBA0_SEC
$ define/system RBA0MODE “OFFLINE”
```

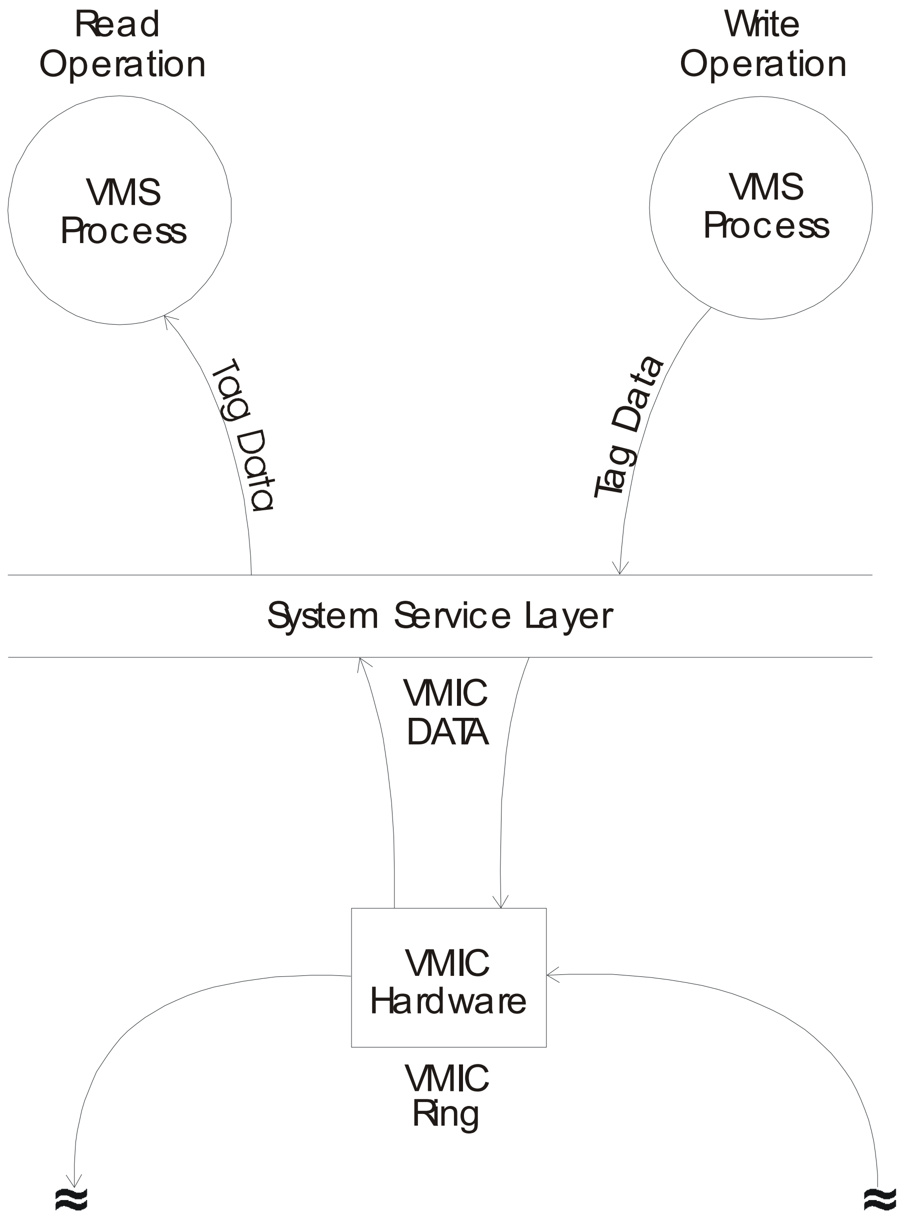


Figure 3-2 NORMAL MODE

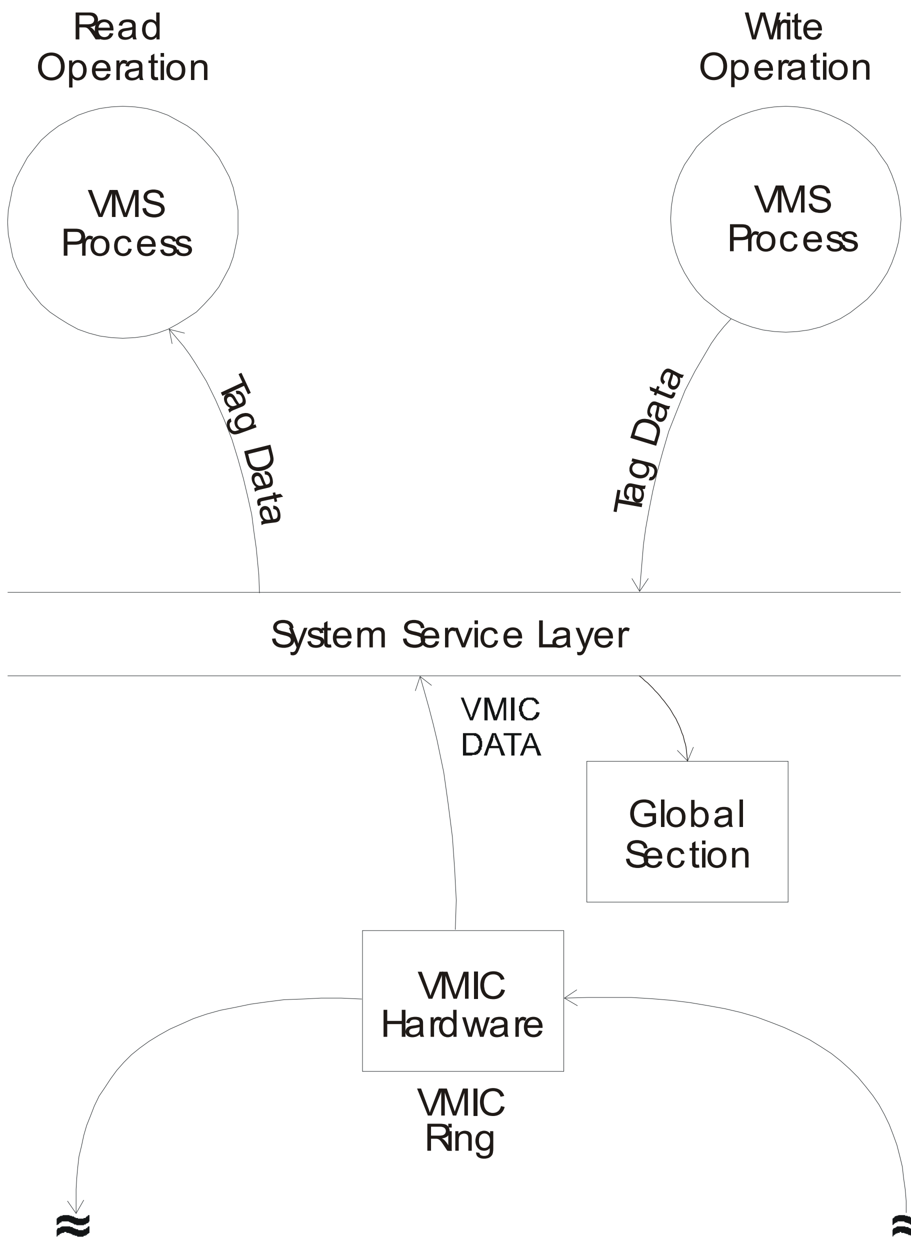


Figure 3-3 READONLY MODE

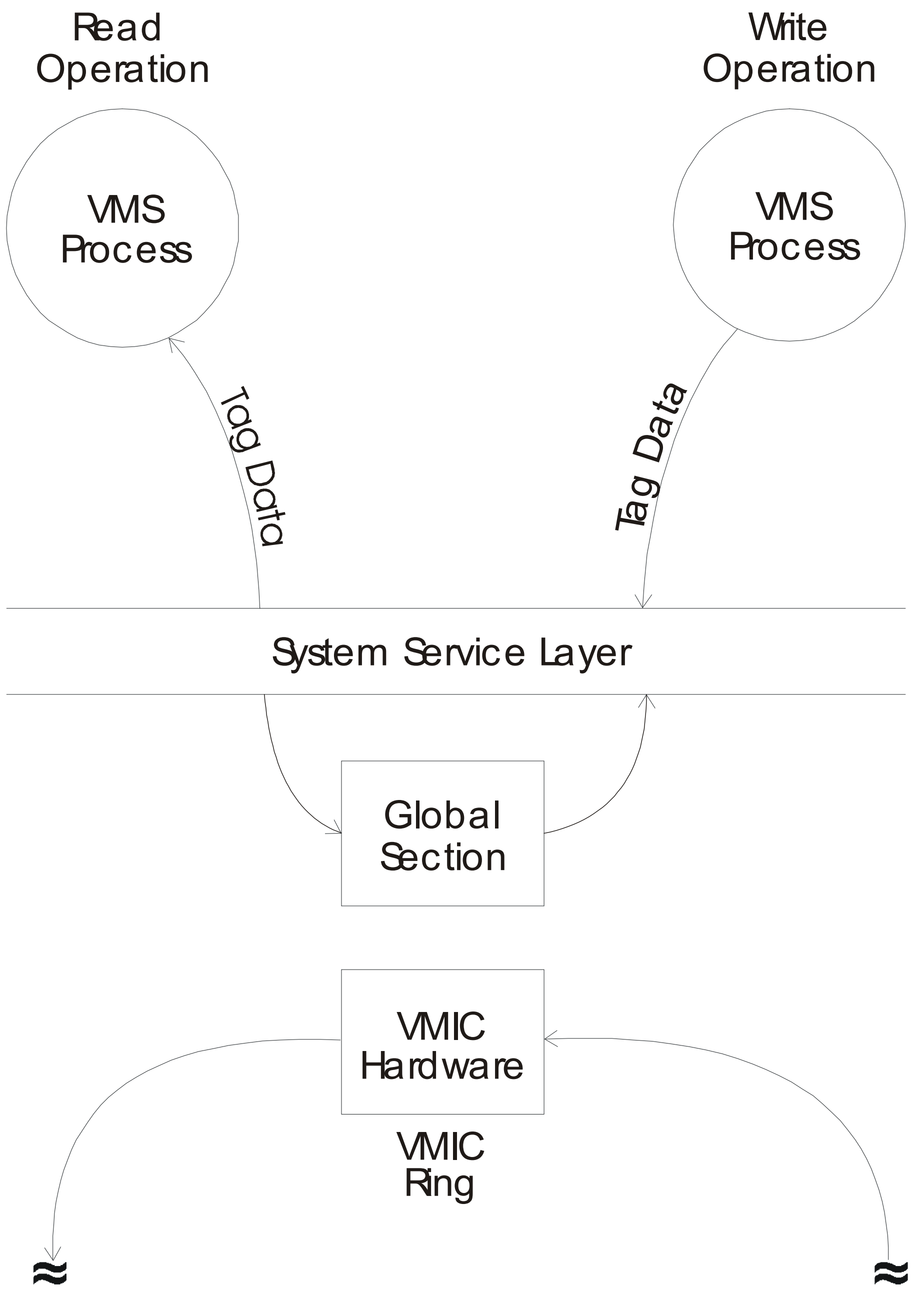


Figure 3-4 OFFLINE MODE

3.2 TAG DATABASE (MASTER CONFIGURATION FILE)

If desired, the memory locations within the Reflective Memory can be addressed symbolically, referred to as TAGS. Each tag has a name, data type, size, and a description. Each TAG has a single source (written by only one node). All other nodes may read the information, but only the source node may write to the tag. The PSTCFG process creates and maintains updates configuration file (ipd_prod:rfmmcf.dat).

3.2.1 TAG DATA TYPES

The Logan implementation of IPD supports 12 data types. A brief description of each data type is included in this section. The user must keep in mind that the minimum size value for the Logan Hot Mill project is 4 bytes. If a data type is shown as less than 4 bytes, the remaining bytes are allocated for the data type but are unused. Data types that are defined as non-longword aligned will always be expanded to be longword aligned. The user should use datatypes that are compatible for all systems. The definition of the datatype is used for debugging purposes. Only the starting address and length are actually used when reading or writing to reflective memory.

DATATYPE	SIZE (bytes)	DESCRIPTION
TIMESTAMP	8	An OpenVMS formatted time value in binary form.
FLOAT	4	A 4 byte floating point value. For the Logan implementation, this is IEEE formatted.
INTEGER	4	A 32 bit signed integer value.
DISCRETE	4	A 32 bit unsigned value used to represent an ON/OFF type of status. The least significant bit is used for this purpose. Its contents may be derived from a single bit in a reflective memory longword.
SHORT	2	A 16 bit signed integer value.
CHAR	1	A single byte signed value.
UINTEGER	4	A 32 bit unsigned integer value.
USHORT	2	A 16 bit unsigned integer value.
UCHAR	1	A single byte unsigned value.
DOUBLE	8	An 8 byte floating point value. For the Logan implementation, this is IEEE formatted.
TEXT	n	A text string of "n" bytes. The size is defined during generation of the database. If the text string is not longword aligned, its size is expanded to be longword aligned. Bytes in excess of the specified text size are reserved, but are unused.
STRUCT	n	A user defined structure tag. The size is defined during generation of the database. If the structure is not longword aligned, its size is expanded to be longword aligned. Bytes in excess of the specified structure size are reserved, but are unused. The database has no knowledge of the fields within the structure and treats the tag as a region of "n" bytes.

Table 3-1 IPD Data Types

3.2.2 RFMOpen Function

The RFMOpen function is provided as the mechanism through, which an application gains access to the VMIC Reflective memory, and the associated tag information. This function must be the first of the IPD functions called in all cases. Calling the RFMOpen function temporarily places the application in EXEC mode. While in EXEC mode, an RFMHandle structure is created. The handle structure is used as a container for information related to the application's request for access to the reflective memory information. It contains a channel number to the VMIC device, if operating in NORMAL or READONLY mode, as well as pointer information to the memory resident group global section created for the MCF copy in memory. Since the handle structure is created while in EXEC mode, it is not accessible by the application once control is returned to the application in USER mode. Once a successful call to RFMOpen has been made, the other five functions may be used. The "C" and "FORTRAN" prototypes for the RFMOpen function are shown below. All the RFM functions have been designed for ease of use in the FORTRAN environment. The passing mechanism shown in the "C" prototype is consistent with the default FORTRAN passing mechanism.

```
int RFMOpen(       struct dsc$descriptor  *pDevName,
              IPDRFM_HANDLE  **pHandle,
              Int *pEFN,
              Int MaxTags);

Integer*4 RFMOpen(Character*10     DevName,
              Integer*4 Handle,
              Integer*4 EFN,
              Integer*4 MaxTags)
```

DevName –

Usage:	*Device name*
Type:	*character-coded text string*
Access:	*read-only*
Mechanism:	*by descriptor*

The address of a character string descriptor pointing to a name string for the VMIC device to be used in reflective memory accesses. Typically this will be RMA0 in a single controller system. If multiple reflective memory cards are present in a system, the device name may be RBA0, RBB0, or RBC0. The VMIC device driver currently supports a maximum of four devices per system, but will typically be limited by the number of slots available in the PCI or PCIX bus.

Handle –

Usage:	*The address of a variable to receive the EXEC mode pointer to the Handle structure.*
Type:	*Longword*

Access: *Write*
Mechanism: *By reference*

The address of a variable that will receive the pointer to the EXEC mode handle structure. This variable will be used in all subsequent calls to the interface functions.

EFN –

Usage: *Event flag number*
Type: *Longword*
Access: *Read-only*
Mechanism: *By reference*

An event flag number to be used internally by the RFM function calls. The event flag should be obtained by use of the LIB$GET_EF call to insure that the event flag is unique in the context of the application.

MAXTAGS –

Usage: *Maximum number of tags caller will reference*
Type: *Longword*
Access: *Read-only*
Mechanism: *By value*

Space must be allocated for each tag that the caller intends to read or write to. This parameter should be specified as zero for the default value or to the number of tags the user intends to access. The system service stores the data type, VMIC address, data length, and if it is an output or output tag in protected address space allocated in executive mode and pointed to by the returned handle variable.

3.2.2.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

IPDM_SS_MODE_NOT_SUPPORTED – *Indicates that the logical name IPDRFM_MODE is either not defined, or its translation string identifies a mode that is not currently supported. Re-define the logical with a translation name of NORMAL, READONLY, or OFFLINE.*

SS$_ACCVIO –*Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*

IPDM_SS_DEV_NAME_TOO_LONG *– Indicates that the user has passed a device name string which is larger than the maximum allowed device name of ten characters. Change the device name string to a length of less than ten characters. It should be RMA0, RMB0, or RMC0.*

IPDM_SS_DEV_LETTER_BAD *– Indicates that the second character of the device name is other than A, B, or C. Correct the device name and try again.*

IPDM_SS_OFFSET_NOT_MUL4 *– The offset value provided is not a multiple of four. The offset and total size provided must be evenly divisible by four. Please correct the value and try again.*

IPDM_SS_OFFSET_NOT_4BYTES *– The specified size must be at least 4 bytes. Please correct the size value and try again.*

LIB$ and SS$ RETURN CODES *– The RFM functions may also return standard OpenVMS system service and library routine status values. Please refer to the OpenVMS manuals for these codes and the appropriate corrective action.*

3.2.3 RFMGetReference Function

The RFMGetReference function is used to retrieve information for the tags, which the application will be using during its processing. The information for each specified tag is retrieved from the configuration file. The information for each tag is stored in a structure supplied by the user for this call. The structure has a specific format that must be strictly adhered to. Each tag must have a structure such as this, defined for it. These structures may then be included in a container structure for ease of access. The format of the structure, called APITAG, is shown in the table below. Each tag structure may contain additional information after the items shown below, but the total bytes per tag should be long word aligned. Multiple calls to RFMGetReference may be used.

FIELD NAME	TYPE	DESCRIPTION
TagName	Character*32	Name of the TAG
pReference	Integer*4	Address of the TAGDEF record for the specified tag in the tag definition area of the memory resident MCF. This value is returned by the call to RFMOpen.
intType	Integer*4	The IPD data type for the specified tag. This value is returned by the call to RFMOpen.
intOffset	Integer*4	The offset in reflective memory space for the specified tag. This value is returned by the call to RFMOpen.
intSize	Integer*4	The size of the region following this field in the structure, which will be used to store a local copy of the tag value. This value is supplied by the caller of RFMOpen.
User defined	User defined	This space is reserved by the user specification, to contain a copy of the tag's value. When used with an RFMRead operation, this area will receive the current value for the specified tag. When used in conjunction with the RFMWrite function, the area will contain the new value to be written to the tag. The size of this area is defined by the value in the intSize field above.

Table 3-2 APITAG Structure

The minimum size for an APITAG structure is 52 bytes. This is because the minimum size of a tag value in reflective memory is 4 bytes. The size of the fields above the User

Defined portion of the APITAG structure is 48 bytes. This plus the minimum tag value size of 4 renders the minimum size structure of 52 bytes.

The "C" and "FORTRAN" prototypes for the RFMGetReference function are shown below. The functions have been designed for ease of use in the FORTRAN environment. The passing mechanism shown in the "C" prototype is consistent with the default FORTRAN passing mechanism.

```
int RFMGetReference( IPDRFM_HANDLE    *pHandle,
                     APITAG           *Tags,
                     int              *TagCnt,
                     int              *pStrucSize);

Integer*4 RFMGetReference(  Integer*4   Handle,
                            Integer*4   Tags,
                            Integer*4   TagCnt,
                            Integer*4   StrucSize)
```

Handle –

Usage:	*A pointer to the EXEC mode Handle structure.*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

Tags –

Usage:	*Address of APITAG structure for tag to be referenced.*
Type:	*Longword*
Access:	*Write*
Mechanism:	*By reference*

The Tags parameter contains the address of a single APITAG structure or a contiguous block of APITAG structures

TagCnt –

Usage:	*Number of tags in the Tags structure.*
Type:	*Longword*
Access	*Read-only*
Mechanism:	*By reference*

This is the address of a variable that contains the number of tags within the Tags structure. If more than a single tag is provided in the Tags structure, they must be contiguous within the containing structure.

Size –

Usage:	*Size of the entire Tag structure provided*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

The address of a variable containing the size, in bytes, of the entire Tag structure provided. For a single tag, this value would be the size of the fixed portion of an APITAG structure, plus the user supplied size value of the tag data. This must be provided in order for the system service code to determine if the user's entire APITAG structure is accessible from EXEC mode.

3.2.3.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

IPDM_BADTAGCOUNT – *Indicates that the user has requested an invalid number of tags. The number of tags must be greater than 0 but less than 500.*

IPDM_APIHDLCORRUPT – *Indicates that the handle pointer passed by the user is pointing to a structure, which has an invalid structure identifier in it. The IPDRFM_HANDLE structure has a specific identification stored within it following a successful RFMOpen call. Please make sure that the application has called RFMOpen prior to passing the handle to the RFMGetReference function.*

IPDM_TAGELEMENTMODULUS – *Indicates that the size of the tag passed to the function is not an even multiple of 4 bytes. Please correct the size and try again.*

IPDM_TAGELEMENTSIZE – *Is an indication that the size of the requested tag is larger than the allowable maximum of 1024 bytes. Correct the invalid size and try the call again.*

IPDM_TAGELEMENT – *Indicates that the size of the element given is smaller than the minimum possible size. The fixed portion of an APITAG structure is 4 bytes8. This plus the size of the smallest possible tag, 4 bytes, results in a minimum size for an APITAG of 52 bytes.*

IPDM_NOTFOUND – *The specified tag name was not found in the memory resident copy of the MCF. Please correct the tag name and try again. If the APITAG structure contains more than a single tag, you must use the RFMExtendedStatus function to determine the failing tag.*

IPDM_SS_MAP_OUT_OF_RANGE – *Indicates that the current reflective memory limit values, set by logical names, exceeds physical limits of the reflective memory card present. Redefine the limit logical names for your environment and try again.*

IPDM_SS_WRTLIM_VIOLATION – *Indicates that one of the tags defined as an output tag in the current operation would result in a write to an area of reflective memory, which is not within the local nodes writable area. Check the write limit logical name definitions and the tags to be written in the APITAG structure.*

SS$_ACCVIO –*Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*

Example programs, written in FORTRAN, can be found in the IPD_EXA directory of the IPD directory tree.

3.2.4 RFMRead

The RFMRead function reads raw reflective memory data into the user's buffer. The source of the data read is dependent upon the current operating mode (e.g., NORMAL, READONLY, or GHOST).

The prototypes for the RFMRead function in both "C" and FORTRAN are shown below.

```
int RFMRead( IPDRFM_HANDLE    *pHandle,
                     void     *Buffer,
                     int      Offset,
                     int      Bytecnt);

Integer*4 RFMRead(  Integer*4     Handle,
                     Integer*4     Buffer,
                     Integer*4     Offset,
                     Integer*4     ByteCnt)
```

Handle –
Usage: *A pointer to the EXEC mode Handle structure.*

Type: *Longword*
Access: *Read-only*
Mechanism: *By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

Buffer –
Usage: *Address of target buffer*
Type: *Longword*
Access: *Write*
Mechanism: *By reference*

The buffer should be on a long word boundry.

Offset –
Usage: *Reflective memory offset.*
Type: *Longword*
Access *Read-only*
Mechanism: *By value*

This is the offset or address on the reflective memory board of the data to be returned.

Size –
Usage: *Number of bytes to return starting at the passed offset*
Type: *Longword*
Access: *Read-only*
Mechanism: *By value*

The number of bytes to be returned. This value should be a modulus of four bytes.

3.2.4.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

SS$_ACCVIO – *Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*

IPDM_APIHDLCORRUPT – *Indicates that the handle pointer passed by the user is pointing to a structure that has an invalid structure identifier in it. The IPDRFM_HANDLE structure has a specific identification stored within it following a successful RFMOpen call. Please make sure that the application has called RFMOpen prior to passing the handle to the RFMGetReference function.*

The function call may also return any of the return codes defined by the device driver.

3.2.5 RFMReadTags

The RFMReadTags function reads the value of a tag or a list of tags, into the calling application's APITAG structure. The source of the information read is dependant upon the current operating mode (e.g., NORMAL, READONLY, or GHOST). The value or values can then be accessed using standard "C" or FORTRAN record access syntax. For example, if the application defined an APITAG structure by the name of L2_TEST_TAG, the application could access the data for the requested tag in one of two ways depending on the data type of the tag. If the data type is not STRUCT, the tag's value is accessed in the following way.

Local_variable_value = L2_TEST_TAG.V

If the tag happened to be of type STRUCT, then the user would access the fields of the user defined structure data using the following syntax.

Local_variable = L2_TEST_TAG.V.field_val_1

This example assumed that the user-defined structure in reflective memory contains a field that is named field_val_1.

The prototypes for the RFMReadTags function in both "C" and FORTRAN are shown below.

```
int RFMReadTags(      IPDRFM_HANDLE     *pHandle,
                          APITAG                *Tags,
                          int                   *TagCnt,
                          int                   *pStrucSize);

Integer*4 RFMReadTags(    Integer*4     Handle,
                          Integer*4     Tags,
                          Integer*4     TagCnt,
                          Integer*4     StrucSize)
```

Handle –
Usage: *A pointer to the EXEC mode Handle structure.*

Type: *Longword*

Access:	*Read-only*
Mechanism:	*By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

Tags –

Usage:	*Address of APITAG structure for tag to be referenced*
Type:	*Longword*
Access:	*Write*
Mechanism:	*By reference*

The Tags parameter contains the address of a single APITAG structure or a contiguous block of APITAG structures

TagCnt –

Usage:	*Number of tags in the Tags structure.*
Type:	*Longword*
Access	*Read-only*
Mechanism:	*By reference*

This is the address of a variable that contains the number of tags within the Tags structure. If more than a single tag is provided in the Tags structure, they must be contiguous within the containing structure.

Size –

Usage:	*Size of the entire Tag structure provided*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

3.2.5.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

SS$_ACCVIO – *Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*

IPDM_BADTAGCOUNT – *Indicates that the user has requested an invalid number of tags. The number of tags must be greater than 0 but less than 500.*

IPDM_APIHDLCORRUPT – *Indicates that the handle pointer passed by the user is pointing to a structure that has an invalid structure identifier in it. The IPDRFM_HANDLE structure has a specific identification stored within it following a successful RFMOpen call. Please make sure that the application has called RFMOpen prior to passing the handle to the RFMGetReference function.*

IPDM_TAGELEMENTMODULUS – *Indicates that the size of the tag passed to the function is not an even multiple of 4 bytes. Please correct the size and try again.*

> ***IPDM_TAGELEMENTSIZE –*** *Is an indication that the size of the requested tag is larger than the allowable maximum of 1024 bytes. Correct the invalid size and try the call again.*
>
> ***IPDM_TAGSTRUCTTOOSMALL –*** *Indicates that the size of the element given is smaller than the minimum possible size. The fixed portion of an APITAG structure is 48. This size plus the size of the smallest possible tag, 4 bytes, results in a minimum size for an APITAG of 52 bytes.*
>
> ***IPDM_NOTFOUND –*** *The specified tag name was not found in the memory resident copy of the MCF. Please correct the tag name and try again. If the APITAG structure contains more than a single tag, you must use the RFMExtendedStatus function to determine the failing tag.*

The function call may also return any of the return codes defined by the VMIC system service routines. Please check the VMIC documentation for these codes.

3.2.6 RFMWrite

The RFMWrite function writes raw data from the user's buffer into reflective memory. The source of the data written is dependent upon the current operating mode (e.g., NORMAL, READONLY, or GHOST).

The prototypes for the RFMWrite function in both "C" and FORTRAN are shown below.

```
int RFMRead(  IPDRFM_HANDLE      *pHandle,
                          void       *Buffer,
                          int        Offset,
                          int        Bytecnt);

Integer*4 RFMWrite(   Integer*4        Handle,
                          Integer*4        Buffer,
                          Integer*4        Offset,
                          Integer*4        ByteCnt)
```

Handle –

Usage:	*A pointer to the EXEC mode Handle structure.*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

Buffer –

Usage:	*Address of target buffer*
Type:	*Longword*
Access:	*Write*
Mechanism:	*By reference*

The buffer should be on a long word boundry.

Offset –

Usage:	*Reflective memory offset.*
Type:	*Longword*
Access	*Read-only*
Mechanism:	*By value*

This is the offset or address on the reflective memory board of the data to be returned.

Size –

Usage:	*Number of bytes to write starting at the passed offset*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By value*

The number of bytes to be returned. This value should be a modulus of four bytes.

3.2.6.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

> ***SS$_ACCVIO –*** *Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*
>
> ***IPDM_APIHDLCORRUPT –*** *Indicates that the handle pointer passed by the user is pointing to a structure that has an invalid structure identifier in it. The IPDRFM_HANDLE structure has a specific identification stored within it following a successful RFMOpen call. Please make sure that the application has called RFMOpen prior to passing the handle to the RFMGetReference function.*

The function call may also return any of the return codes defined by the device driver.

3.2.7 RFMWriteTags

The RFMWriteTags function writes the value of a tag or list of tags, to reflective memory, or associated memory global section, depending on the current operating mode. Using standard "C" or FORTRAN record access syntax sets the value or values for the tags. For example, if the application defined an APITAG structure by the name of L2_TEST_TAG, the application could set the data for the requested tag in one of two ways depending on the data type of the tag. If the data type is not STRUCT, the tag's value is accessed in the following way.

L2_TEST_TAG.V = new_value

If the tag happened to be of type STRUCT, then the user would access the fields of the user defined structure data using the following syntax.

L2_TEST_TAG.V.field_val_1 = new_value

This example assumed that the user-defined structure in reflective memory contains a field that is named field_val_1.

The prototypes for the RFMReadTags function in both “C” and FORTRAN are shown below.

```
int RFMWriteTags(          IPDRFM_HANDLE     *pHandle,
                           APITAG            *Tags,
                           int               *TagCnt,
                           int               *pStrucSize);

Integer*4 RFMWriteTags(    Integer*4    Handle,
                           Integer*4    Tags,
                           Integer*4    TagCnt,
                           Integer*4    StrucSize)
```

Handle –

Usage:	*A pointer to the EXEC mode Handle structure.*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

The address of a variable, which contains the pointer to the EXEC mode, handle structure.

Tags –

Usage:	*Address of APITAG structure for tag to be referenced*
Type:	*Longword*
Access:	*Read only*
Mechanism:	*By reference*

The Tags parameter contains the address of a single APITAG structure or a contiguous block of APITAG structures

TagCnt –

Usage:	*Number of tags in the Tags structure.*
Type:	*Longword*
Access	*Read-only*
Mechanism:	*By reference*

The address of a variable that contains the number of tags within the Tags structure. If more than a single tag is provided in the Tags structure, they must be contiguous within the containing structure.

Size –

Usage:	*Size of the entire Tag structure provided*
Type:	*Longword*
Access:	*Read-only*
Mechanism:	*By reference*

3.2.7.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

> ***SS$_ACCVIO –*** *Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*
>
> ***IPDM_BADTAGCOUNT –*** *Indicates that the user has requested an invalid number of tags. The number of tags must be greater than 0 but less than 500.*
>
> ***IPDM_APIHDLCORRUPT –*** *Indicates that the handle pointer passed by the user is pointing to a structure that has an invalid structure identifier in it. The IPDRFM_HANDLE structure has a specific identification stored within it following a successful RFMOpen call. Please make sure that the application has called RFMOpen prior to passing the handle to the RFMGetReference function.*
>
> ***IPDM_SS_MODE_NOT_SUPPORTED –*** *Indicates that the mode logical name IPDRFM_MODE has been set to a value that is not currently supported. The value must be NORMAL, READONLY, or OFFLINE.*
>
> ***IPDM_TAGELEMENTMODULUS –*** *Indicates that the size of the tag passed to the function is not an even multiple of 4 bytes. Please correct the size and try again.*
>
> ***IPDM_TAGELEMENTSIZE –*** *Is an indication that the size of the requested tag is larger than the allowable maximum of 1024. Correct the invalid size and try the call again.*
>
> ***IPDM_TAGSTRUCTTOOSMALL –*** *Indicates that the size of the element given is smaller than the minimum possible size. The fixed portion of an APITAG structure is 48. This plus the size of the smallest possible tag, 4 bytes, results in a minimum size for an APITAG of 52 bytes.*
>
> ***IPDM_NOTFOUND –*** *The specified tag name was not found in the memory resident copy of the MCF. Please correct the tag name and try again. If the APITAG structure contains more than a single tag, you must use the RFMExtendedStatus function to determine the failing tag.*

The function call may also return any of the return codes defined by the VMIC system service routines. Please check the VMIC documentation for these codes.

3.2.8 RFMExtendedStatus Function

The RFMExtendedStatus function is provided as the mechanism through, which an application gains access to extended status from a prior RFM system service call. A typical use for the call is in the case of an RFMGetReference where the APITAG structure contains multiple tags. In order to determine the identity of the offending tag from the group of tags, the user would call the RFMExtendedStatus routine. The index to the offending tag would be returned to the caller. The same case exists with the RFMReadTags and RFMWriteTags when multiple tags are present in the APITAG structure.

```
int RFMExtendedStatus(IPDRFM_HANDLE    *pHandle,
                      RFM_IOSB         *pIosb);
```

```
Integer*4 RFMExtendedStaus(Integer*4      Handle,
                           Integer*4      Iosb)
```

Handle –
Usage: *The pointer to the EXEC space Handle structure.*
Type: *Longword*
Access: *Read only*
Mechanism: *By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

Iosb –

Usage: *IO Status Buffer*
Type: *Longword Array(2 elements)*
Access: *Write*
Mechanism: *By reference*

An array of two longwords that will receive the extended status code and the index to the offending tag.

3.2.8.1 Function Return Codes

The function may return any of the following return codes. As with all OpenVMS functions, an odd value indicates success, and an even code indicates failure.

> ***SS$_ACCVIO*** *–Indicates that the system service code does not have access to at least one of the parameters passed to it. This may mean that a pointer has not been initialized before calling the system service function.*

3.2.9 RFMClose Function

The RFMClose function is provided as the mechanism through, which an application terminates its access to the reflective memory environment and a value of NULL is placed in the Handle structure pointer variable. An application that intends to exit, need not call this function as normal OpenVMS process rundown will release all resources captured by the process.

int RFMClose(IPDRFM_HANDLE **pHandle)

*Integer*4 RFMClose*(Integer*4 Handle)

Handle –
Usage: *The address of a variable containing the pointer to the EXEC mode Handle structure.*
Type: *Longword*
Access: *Write*
Mechanism: *By reference*

The address of a variable that contains the pointer to the EXEC mode handle structure.

4 PCI-5565 Device Driver

The device driver provides the actual access and control of the GE FANUC PCI-5565 device installed in the PCI or PCIX bus. The driver provides synchronized access to the reflective memory on the card from multiple OpenVMS Processes and multiple CPU cores. The actual driver QIO (OpenVMS Driver system interface) calls are shown in this section. However, the end user should reference the provided RFM system service API (Application Programming Interface) instead of the driver QIO interface when possible. The device driver supports Symmetric Multi-Processing (SMP).

The driver supports up to four devices. The OpenVMS device name for each PCI-5565 is RBA0:, RBB0:, RBC0:, and RBD0:. Currently there is no real software limit in the driver, but bus space and PCI address space may limit the actual number of devices that may be installed in the system.

After the kit is installed (normally installed in vms$common:[ipdproduct]), the command file: RFM_5565_STARTUP.COM is executed which calls the RFM_DRIVER_LOAD.COM command procedure to actually load the driver. The RFM_DRIVER_LOAD command procedure is created during the kit installation or may be created using the FINDPCI utility.

The different QIO function calls are summarized here and further documented in with their parameters and possible error codes in each subsequent section. The function codes shown in parenthesis are from OpenVMS $IODEF and are chosen to be similar to existing OpenVMS device driver function codes. The user should reference the OpenVMS System Service Manual for a description of the SYS$QIO and SYS$QIOW system service calls RFM_DRIVER_LOAD.COM. The RFM_DRIVER_LOAD.COM command procedure is created during the kit installation procedure by the FINDPCI utility.

- Write Virtual Block(IO$_WRITEVBLK)- Write a block of memory to the reflective memory ring (buffered I/O)
- Write Physical Block(IO$_WRITEPBLK)- Write a block of memory to the reflective memory ring (DMA)
- Write Interrupt Word (IO$_WRITEHEAD)- Write interrupt data to a node on the reflective memory ring.
- Read Virtual Block(IO$_READVBLK)- Read a block of memory from the reflective memory ring (buffered I/O)

- Read Physical Block(IO$_READPBLK- Read a block of memory from the reflective memory ring (DMA)
- Read Network FIFO(IO$_READRCT)- Read one of the four network FIFOs
- Read Init Node FIFO(IO$_READHEAD)- Read FIFO of nodes that have joined reflective memory ring.
- Read Control Buffer(IO$_READTRACKD)- Read the auxiliary controller structure (CBF).
- Refresh Memory to ring(IO$_READPBLK + IO$M_REQ_MINICOPY)- Function to read and write local reflective memory for nodes that have recently joined the reflective memory ring.
- Set Endian(IO$_INITIALIZE + IO$M_SWAP)- Set the Endian mode
- Set Memory Write Range(IO$_INITIALIZE + IO$M_UPDATE_MAP)- Set starting address to allow writes to and length of memory for this node. If left as zero, no write validations are performed.
- Enable Network Interrupts(IO$_INITIALIZE + IO$M_INTERRUPT)
- Register for Init Node wakeup(IO$_INITIALIZE + IO$M_INTERRUPT+IO$M_BOOTING)- Register for event flag to be set when FIFO has data (e.g., new node has joined ring)
- Rogue Master Enable(IO$_INITIALIZE + IO$M_SET_MODEM+IO$M_PURGE)- Enables this node to detect rogue packets.
- Loopback Mode Enable/Disable(IO$_INITIALIZE + IO$M_SET_MODEM+IO$M_LOOP)
- Set Dark on Dark mode (IO$_INITIALIZE + IO$M_SET_MODEM+IO$M_UNLOOP)
- IO$_IOCTLV- Unix IOCTL functions
- IO$_IOSENSEMODE- Standard sense mode functions

All reads and writes to the reflective memory ring are 32 bit longwords. The driver supports non-aligned data (user buffer and reflective memory addresses) for the buffered transfers. However, if possible, all buffers and byte counts should be 4 byte aligned and have a 4 byte length modulus. This is a requirement for the DMA functions.

4.1 Write Virtual Block (IO$_WRITEVBLK)

This function will write the user's buffer to the reflective memory address passed as the P3 parameter. Because of the trade off of locking a buffer down in memory and the typical small size of the transfer, this function is implemented using buffered I/O. Any writes to odd boundaries in reflective memory results in a read of the lower four-byte modulus address longword (4 byte) with a merge of the data. Users should develop applications that attempt to align data in reflective memory on longword boundaries. Data buffers do not need to be longword aligned, but efficiencies will be achieved if they are.

QIO Parameters	
Parameter	**Description**
1	User buffer to be written to reflective memory
2	Byte count to transfer
3	Starting byte address in reflective memory

4.2 Write Physical Block(IO$_WRITEPBLK)

This function will write the user's buffer to the reflective memory address passed as the P3 parameter. This function is designed for larger transfers and will execute without the host processor intervention. The user buffer and target reflective memory addresses must be longword aligned and the byte count must be a modulus of four bytes. The Maximum transfer size is 64K bytes limited by both OpenVMS and the PCI-5565. To increase throughput, the user buffer can be locked down in memory (see sys$lkwset).

QIO Parameters	
Parameter	**Description**
1	Longword aligned user buffer to be written to reflective memory
2	Byte count to transferred (modulus of 4 bytes)
3	Longword aligned starting byte address in reflective memory

4.3 Read Virtual Block(IO$_READVBLK)

This function will read into the user's buffer from the reflective memory address passed as the P3 parameter. Because of the trade off of locking a buffer down in memory and the typical small size of the transfer, this function is implemented using buffered I/O. Any reads from odd boundaries in reflective memory results in a read of the lower four-byte modulus address longword (4 byte). Users should develop applications that attempt to align data in reflective memory on longword boundaries. Data buffers do not need to be longword aligned, but efficiencies will be achieved if they are.

QIO Parameters

Parameter	Description
1	User buffer to be read from reflective memory
2	Byte count to transfer
3	Starting byte address in reflective memory

4.4 Read Physical Block(IO$_READPBLK)

This function will read into the user's buffer from the reflective memory address passed as the P3 parameter. This function is designed for larger transfers and will execute without the host processor intervention. The user buffer and target reflective memory addresses must be longword aligned and the byte count must be a modulus of four bytes. The Maximum transfer size is 64K bytes limited by both OpenVMS and the PCI-5565. To increase throughput, the user buffer can be locked down in memory (see sys$lkwset).

QIO Parameters	
Parameter	**Description**
1	Longword aligned user buffer to be read from reflective memory
2	Byte count to transferred (modulus of 4 bytes)
3	Longword aligned starting byte address in reflective memory

4.5 Read Network FIFO(IO$_READRCT)

The PCI-5565 device has four 128 entry FIFOs that can be enabled individually. Data is written to these FIFOs based on the interrupt type passed in the message. This can be from one to four. The sending node id and the data is placed into the selected FIFO. This function allows the user to read the particular FIFO. The (IO$_INTIALIZE + IO$M_READATTN) must be enabled for the particular FIFO.

QIO Parameters	
Parameter	**Description**
1	Longword aligned user buffer to be read from reflective memory. Two longwords are returned for each entry. The first longword contains the sender node and the second contains the sender’s data.
2	Byte count to transferred (modulus of 8 bytes)
3	FIFO number (1,2, 3, or 4)

4.6 Read Init Node FIFO(IO$_READHEAD)

The driver maintains a bit map of nodes (if function is enabled) that have recently joined the reflective memory ring (eight long words). This function will return this bit map and reset it. The recognition of these events must be enabled by the using the IO$_INTIALIZE + IO$M_BOOTING function.

QIO Parameters	
Parameter	**Description**
1	Eight longword user buffer to be read from reflective memory. The reflective memory ring support up to 256 nodes.
2	Byte count to transferred (32 bytes)

4.7 Read Control Buffer(IO$_READTRACKD)

This function is a diagnostic function that returns the auxiliary controller structure (CBF) from the driver for the particular device. The definition of this structure is contained in the driver definition header file (RFM_5565_DEF.H).

QIO Parameters	
Parameter	**Description**
1	User buffer to be receive the CBF from the driver. Use a variable of type: CBF as defined in RFM_5565_DEF.h.
2	Byte count to transferred from CBF (use size of (CBF))

4.8 Set Endian(IO$_INTIALIZE)

This will change the current Endian setting for the PCI-5565 device. The single parameter should be either 0= Little Endian (default power up state of PCI-5565) and 1= Big Endian. This effects how bytes are written to the reflective memory by the host and therefore which byte is sent out first on the ring. In addition, the data being written to reflective memory by incoming packets can also be changed. This has no effect on the interrupt FIFOs.

QIO Parameters	
Parameter	**Description**
1	Host writes to reflective memory Endian Setting (0= Little, 1= Big)
2	Slave writes to reflective memory Endian Setting (0= Little, 1= Big)

4.9 Set Memory Write Range(IO$_INTIALIZE)

The PCI-5565 allows the host node to write into any location within reflective memory. This function allows the user to specify a range of memory partitions (up to three) that will be checked before a write can occur. By default, the full range is available when the driver loads. This enables the validation check of both buffered and DMA write transfers to reflective memory for the host node. All writes must be contained in one of the three valid writeable partitions unless IO$M_TRUSTED modifier is set in the write function. The IO$M_TRUSTED modifier is used for simulation purposes and disables this validation check. The partitions should be specified in ascending beginning reflective memory address. Depending on the memory installed, the upper limit is:

4000000 Hex 67,108,864 Decimal for 64 MB option
8000000 Hex 134,217,728 Decimal for 128 MB option

The actual amount mapped can be less than available on the card if there are insufficient SPTEs to map the PCI address space. The utility DEVDMP can be used after the driver is loaded to identify how much of the memory is actually available to the system. The following snippet from the output of DEVDMP is shown below (this example shows all of the memory available, it was executed on an Itanium):

```
0x0208x ucb$l_card_size   0x04000000 (64 megs) mapped
                          0x04000000 64Mb total card memory
```

QIO Parameters

Parameter	Description
1	Start of writeable partition one in reflective memory
2	Length of writeable partition in bytes
3	Start of writeable partition two in reflective memory
4	Length of writeable partition in bytes
5	Start of writeable partition three in reflective memory
6	Length of writeable partition in bytes

4.10 Enable Network Interrupts(IO$_INTIALIZE)

This will enable one of the network interrupt data FIFOs or disable all of the interrupt data FIFOs. The function would need to be called once for each of the interrupt data types. The function allows the ability to have the calling process have a local event flag set whenever an entry is placed into the particular FIFO. The FIFO can then be read.

QIO Parameters

Parameter	Description
1	FIFO number (1,2,3, or 4). A zero will disable all interrupts.
2	Optional event flag, zero indicates no event flag is to be set

4.11 Register for Init Node wakeup(IO$_INTIALIZE)

The PCI-5565 can detect when a new node joins the reflective memory ring. This function enables this recognition and optionally allows for the setting of the process's local event flag when this event occurs on the ring. When a node is detected, the associated bit in the init-node bit array is set. The IO$_READHEAD function can be used to read the eight longword bit map.

QIO Parameters	
Parameter	**Description**
1	1= enable, 0= disable
2	Optional event flag, zero indicates no event flag is to be set

4.12 Refresh Memory to ring

This function is used to refresh the local memory owned (writeable) by the local node (see the IO$_READPBLK + IO$M_REQ_MINICOPY). This function will simply read each memory partition and write the data back forcing a ring update. This is typically used after a new node joins the reflective memory ring. The user buffer must be large enough for the largest partition defined for the host's writeable regions. Additionally, the writeable regions must be setup before use of this function. DMA is used first to read each partition and then to write it. This method reduces CPU overhead and the DMA engine automatically throttles the writes to the transmit FIFO. The driver also ensures that the refresh is completed before any other updates to the reflective memory can occur ensuring an atomic refresh.

QIO Parameters	
Parameter	**Description**
1	Longword aligned user buffer to be read/written from reflective memory
2	Byte count of largest writeable partition

4.13 Rogue Master Enable(IO$_INTIALIZE)

This function enables this node as one of the two rogue packet detectors and destroyers. See the PCI-5565 manual for further information.

QIO Parameters	
Parameter	**Description**
1	1= enable, 0= disable
2	Rogue 1 or Rogue 2

4.14 Loopback Mode Enable/Disable(IO$_INTIALIZE)

Enable Loop back mode of the PCI-5565. See PCI-5565 for further documentation.

QIO Parameters	
Parameter	**Description**
1	1= enable, 0= disable

4.15 Set Dark on Dark mode (IO$_INITIALIZE)

Turn off the fiber transmitter when no signal is detected on the fiber receiver.

QIO Parameters	
Parameter	**Description**
1	1= enable, 0= disable

4.16 Error Codes

The PCI-5565 device driver returns status in conformance to OpenVMS standards. The status can be returned via the status of the SYS$QIO function call if the request is not valid or it can be returned in the I/O status quad word. The PCI-5565 device driver uses the OpenVMS symbolic codes defined in $ssdef mapped to the definitions for this driver.

Symbolic Status Code	Description
SS$_NORMAL	Normal completion
SS$_ABORT	I/O was cancelled
SS$TIMEOUT	Device timeout
SS$_NOSLOT	Reflective memory address is not in range of the PCI-5565 currently installed. Validate parameter is passed by value.
SS$_ILLBLKNUM	Buffered I/O size exceeds reasonability. The size exceeds the constant DMA_THRESHOLD. Validate parameter is passed by value or consider using DMA.
SS$_BADPARAM	QIO parameter is in error, zero or negative byte count,
SS$_BUFBYTALI	
SS$_IVBUFLEN	
SS$_ACCVIO	
SS$_ILLBLKUNM	Writeable partition information is invalid. Start or start plus size exceeds PCI-5565 available memory, or length less than zero.
SS$_ILLIOFUNC	Illegal function or function modifier entered.

4.17 Device Loading

The VMSINSTAL kit scans assume the PCI-5565 devices have been installed. The utility FINDPCI examines the results of the POST (Power On and Self Test) stored in the configuration data within the OpenVMS data tables. It then creates the SYSMAN commands to actually load the driver. The FINDPCI utility can be ran at anytime if a second PCI-5565 is added.

The driver uses certain resources that can become exhausted and will not allow the PCI-5565 to be useable. The only mechanism that is available to the driver to report any problems is through the use of the device message service (EXE_STD$SNDEVMSG facility). The messages available limited and this driver uses the message codes for things not necessarily associated with OpenVMS's intended meaning. The following error messages and their meaning with respect to loading the device driver are listed below. The symbolic value comes from MSGDEF.

Symbolic	**OpenVms Device Message**	**PCI-5565 Driver meaning**
MSG$_DEVONLIN	Device RBA0: is online	Normal operation, device is ready. Written after device was found and configuration and mapping of device was successful.
MSG$_MVCOMPLETE	Mount verification has completed for device RBA0:	Normal operation, device controller is ready
MSG$_SHACHASTA	RBA0: shadow set has changed state.	Normal operation, unit on controller is ready
MSG$_DEVOFFLIN		
MSG$_DEVOFFLINX		Error attempting to enable interrupts. Device is offline.
MSG$_DEVWRTLCK		
MSG$_SHARDUCED		

4.18 Device Error and Event Messages

During normal operation, the device driver polls the PCI-5565 (five second interval) to detect any possible problems. When these events are detected, a message is sent using the EXE_STD$SNDEVMSG facility. Any significant event is logged using an OpenVMS device message, which is mapped to the PCI-5565 driver meaning, shown in the following table. Again, the OpenVMS device message is what is printed, and has no meaning with regard to the PCI-5565 driver. In addition to polling the PCI-5565, the status light on the PCI-5565 is toggled each five seconds.

Symbolic	**OpenVms Device Message**	**PCI-5565 Driver meaning**
MSG$_DISMOUNTED	Volume dismounted	Sync was lost
MSG$_SHAMEMFAL	Member failed out of shadow set	Write FIFO almost full
MSG$_MVABORTED	Mount verification has aborted	Receiver no signal present

	for device RBA0:	
MSG$_SHAWROMEM	RBA0: is an incorrect shadow set member volume.	Receiver FIFO full
MSG$_SHAREDZER	RBA0: contains zero working members.	Bad data received

5 Utilities

There are six utility programs supplied as part of this product. Each of these utilities is invoked through the use of a foreign command. A foreign command is a command that is not included as part of the OpenVMS distribution. These commands are created through the use of DCL symbols. The foreign commands for the IPD utilities are shown below and are defined in the command procedure: RFM_SYMBOLS.COM.

```
DEVDMP == “$IPD_PROD:DEVDMP”
GETIT == "$IPD_PROD:GETIT"
PUTIT==  "$IPD_PROD:PUTIT"
PEEK == “$IPD_PROD:RFM_PEEK”
POKE==“$IPD_PROD:RFM_POKE”
RFM== “IPD_PROD:RDM”
PSTCFG == “IPD_PROD:PSTCFG”
```

Each utility is described in the following sections. In addition, some debugging tools are also available: DEVDMP (Dumps device driver data structures) and FINDPCI (browses the adapter tables and displays PCI devices installed in the running system and creates a sysman driver load file for all found PCI-5565 reflective memory cards found).

5.1.1 DEVDMP

The DEVDMP utility is a debugging tool that can help IPACT diagnose any problems with the driver. The syntax of the command is:

$DEVDMP [DeviceName]

DeviceName
The DeviceName is RBA0:, RBB0:, RBC0:, or RBD0:. If not specified, RBA0: is assumed. This utility must be ran from an account that has change mode to kernal privledge. It examines the current running system and extracts a copy of the device data structures and formats them for the user. These data structures are documented in the HP OpenVMS device driver manual and in the header file for these structures (RFM_5565_DEF.H). A sample output is shown below. Some of the data listed that might be of user interest are:

- Actual card size and how much was mapped (ucb$l_card_size)
- PCI configuration information pci$<field name>, see the PCI standard for more details
- Copy of first few registers from reflective memory: cbf_l_rmem
- Device register contents for DMA start and interrupt completion

- Current write partitions set up for this node

Sample Output

```
$ DEVDMP
command line: DEVDMP device_name

 Assuming RBA0:
 Acquiring UCB for device: RBA0:

Device information for: RBA0:

Base memory via DVI$_DEVDEPEND2 : 0x816266c0

0x005c ucb$l_devdepend:      0x00000000
0x0060 ucb$l_devdepnd2:      0x816266c0
0x0064 ucb$l_devdepnd3:      0x00009501 (38145)
0x00b8 ucb$l_boff:           0x00000080
0x00b4 ucb$l_bcnt:           0x00000064
0x00b0 ucb$l_svapte:         0x8162ecc0
0x00f4 ucb$l_msg:            0x00000061
0x00f8 ucb$l_cbf:            0x816113c0
0x00fc ucb$l_pci_page_size:  0x00000020
0x0100 ucb$l_adp:            0x81447600
0x0108 ucb$q_plx_handle:     0x0000000081628ac0
0x0110 ucb$q_rfm_handle:     0x0000000081628b00
0x0118 ucb$q_mem_handle:     0x0000000081628b40
0x0120x ucb$l_card_size      0x03900000 (57 megs) mapped
                             0x04000000 64Mb total card memory
0x016c ucb$l_fault:          0x00000000
0x0170 ucb$l_fault_loc:      0x00000005 (5)
0x0168 ucb$s_crctx:          0x81630fc0

cbf_l_ident:         0xffeaff01
cbf_l_self:          0x816113c0
cbf_q_LoadTime:      13-NOV-2006 17:15:04.0
cbf_l_idb:           0x81628a40
cbf_l_ucb:           0x816266c0
cbf_l_memsize:       0x04000000
cbf_l_adp:           0x81447600
cbf_l_crb:           0x816310c0
cbf_l_TxFIFOFullCtr: 0x00000000
cbf_l_SyncLostCtr:   0x00000000
cbf_l_dma_address:   0x00000000
cbf_l_5565Addr:      0x00000000
cbf_l_ByteCnt:       0x00000000
cbf_l_Dma0Dsc:       0x00000000
cbf_l_itrcsStGo:     0x00000000
cbf_l_DmaMd0Go:      0x00000000
cbf_l_DmaCsrGo:      0x00000000
cbf_l_itrcs_st:      0x0f010100
cbf_l_dmamode0St:    0x00000000
cbf_l_DmaExpected:   0x00000000
cbf_l_DmaTmoCtr:     0x00000000
cbf_l_intrctr:       0x00000000
cbf_l_PciIntCtr:     0x00000000
cbf_l_DmaIntCtr:     0x00000000
cbf_l_LclIntCtr:     0x00000000
cbf_l_UnsolDmaCtr:   0x00000000
cbf_l_itrcs_int:     0x00000000
cbf_l_rfmlcsr1:      0x00000000
cbf_l_rfmlisr:       0x00000000
cbf_l_Dma0Csr_int:   0x00000000
cbf_l_TmoOccurred:   0x00000000
cbf_l_IoDoneCtr:     0x00000000
cbf_l_Lstfunc:       0x00000021
cbf_l_WdTmCtr:       0x0000000a
cbf_l_TdRfmCsr1:     0x80000085
cbf_l_TdRfmLisr:     0x00000000
```

```
cbf_l_StsLed:        0x00000000
cbf_l_SyncSts:       0x00000000
cbf_l_RxvFifoFull:   0x00000000
cbf_l_TxFifoFull:    0x00000000
cbf_l_BadData:       0x00000000
cbf_l_RxSigDet:      0x00000000
Write Partitions not initialized for this node
cbf_l_NewNodes:
                     0x00000000
                     0x00000000
                     0x00000000
                     0x00000000
                     0x00000000
                     0x00000000
                     0x00000000
                     0x00000000
cbf_l_end:           0xffeaff02

lcr_l_lasorr:    0xffffffc0
lcr_l_lasoba:    0xc0000001
lcr_l_marbr:     0x03040000
lcr_b_bigend:    0x00000000
lcr_b_lmisc1:    0x00000005
lcr_b_prot_area: 0x00000030
lcr_b_lmisc2:    0x00000021
lcr_l_rgpcilclr: 0x00000000
lcr_l_rgdescp:   0x00000000
lcr_l_lbrd1:     0x42430143
lcr_l_rgdmpci:   0x00000000
lcr_l_lclbsmpci: 0x00000000
lcr_l_lclbsdm:   0x00000000
lcr_l_pcibsrm:   0x00000000
lcr_l_pcicfgrg:  0x00000000

rtr_l_mbx0:     0x00000000
rtr_l_mbx1:     0x00000000
rtr_l_mbx2:     0x00000000
rtr_l_mbx3:     0x00000000
rtr_l_mbx4:     0x00000000
rtr_l_mbx5:     0x00000000
rtr_l_mbx6:     0x00000000
rtr_l_mbx7:     0x00000000
rtr_l_pcilcldb: 0x00000000
rtr_l_lclpcidb: 0x00000000
rtr_l_itrcs:    0x0f010100
rtr_l_eeprmct:  0x100f767e
rtr_w_vendid:   0x000010b5
rtr_w_devid:    0xffff9656
rtr_w_revid:    0x000000ba
rtr_w_unused:   0x00000000
rtr_l_mbxreg1:  0x00000000
rtr_l_mbxreg2:  0x00000000

dcr_l_dmamode0:  0x00000043
dcr_l_dma0padr:  0x00000000
dcr_l_dma0ladr:  0x00000000
dcr_l_dma0xfr:   0x00000000
dcr_l_dma0dsc:   0x00000000
dcr_l_dmamode1:  0x00000043
dcr_l_dma1padr:  0x00000000
dcr_l_dma1ladr:  0x00000000
dcr_l_dma1xfr:   0x00000000
dcr_l_dma1dsc:   0x00000000
dcr_c_dmacsr0:   0x00000010
dcr_c_dmacsr1:   0x00000010
dcr_w_resrved:   0x00000000
dcr_l_marbr:     0x03040000
dcr_l_dmathr:    0x00000000
dcr_l_dmadac0:   0x00000000
dcr_l_dmadac1:   0x00000000
```

```
rfm_b_brv:   0x00000007
rfm_b_bid:   0x00000065
rfm_b_nid:   0x00000001
rfm_l_lcsr1: 0x80000084
rfm_l_resv3: 0x00000000
rfm_l_lisr:  0x00000000
rfm_l_lier:  0x00000000
rfm_l_ntd:   0x00000000
rfm_b_ntn:   0x00000000
rfm_b_nic:   0x0000000f
rfm_l_isd1:  0x00000000
rfm_b_sid1:  0x00000000
rfm_l_isd2:  0x00000000
rfm_c_sid2:  0x00000000
rfm_l_isd3:  0x00000000
rfm_b_sid3:  0x00000000
rfm_l_initd: 0x00000000
rfm_b_initn: 0x00000000

pci$w_vendor_id:       0x0000114a
pci$w_device_id:       0x00005565
pci$w_command:         0x00000147
pci$w_status:          0x000002b0
pci$b_revision_id:     0x00000001
pci$b_programming_if:  0x00000000
pci$b_sub_class:       0xffffff80
pci$b_base_class:      0x00000002
pci$b_cache_line_size: 0x00000010
pci$b_latency_timer:   0xffffffff
pci$b_header_type:     0x00000000
pci$b_bist:            0x00000000
pci$l_base_address_0:  0x03fb7e00
pci$l_base_address_1:  0x01ffff01
pci$l_base_address_2:  0x03fb7d00
pci$l_base_address_3:  0x04000000
pci$l_base_address_4:  0x00000000
pci$l_base_address_5:  0x00000000
pci$l_cardbus_cis:     0x00000000
pci$w_sub_vndr:        0x000010b5
pci$w_sub_id:          0xffff9656
pci$l_exp_rom_base:    0x00000000
pci$l_reserved_3:      0x00000040
pci$l_reserved_4:      0x00000000
pci$b_intr_line:       0x00000010
pci$b_intr_pin:        0x00000001
pci$b_min_gnt:         0x00000000
pci$b_max_lat:         0x00000000

cbf_l_rmem[0]: 0x00000000
cbf_l_rmem[1]: 0x00000401
cbf_l_rmem[2]: 0x00000802
cbf_l_rmem[3]: 0x00000c03
cbf_l_rmem[4]: 0x00001004
cbf_l_rmem[5]: 0x00001405
cbf_l_rmem[6]: 0x00001806
cbf_l_rmem[7]: 0x00001c07
cbf_l_rmem[8]: 0x00002008
cbf_l_rmem[9]: 0x00002409
cbf_l_rmem[10]: 0x0000280a
cbf_l_rmem[11]: 0x00002c0b
cbf_l_rmem[12]: 0x0000300c
cbf_l_rmem[13]: 0x0000340d
cbf_l_rmem[14]: 0x0000380e
cbf_l_rmem[15]: 0x00003c0f
```

5.1.2 PUTIT

The PUTIT utility is responsible for writing a value to any process variable. This includes a simple type (real, int, byte), or to an array of simple types, and finally, to any element or elements of a structure. **The PUTIT utility can write to any process variable regardless if the process variable is owned by the local system. This utility should be used with caution.** Typically, this utility is used for simulation purposes when the actual owner of the process variable is not controlling the variable.

$PUTIT[/DEBUG/QUIET] Tags Data

Tags
The Tags is the process variable desired to be written. If the process variable is an array, then the element index can be specified using square brackets: "[element number]". The element number always starts with one. If the process variable is an element, then the tag name followed by the desired elements of the structure can be specified, including elements of arrays contained within the structure. Tags can also be a file name that contains a list of elements from the process variable. If more than a single element of a process variable is to be returned, the string must be enclosed in quotes unless the tag string is passed via a data file.

Data
The Data are the corresponding data items for the process variable elements specified in Tags argument. If more than a single value is present, the data items must be separated by commas and the whole string quoted. Additionally, the data can be sourced from a local DCL symbol by specifying a "%symbolname" or from a data file "@datafile". The data file is a simple text file with values for each element of the tag variable. The data file can contain comment lines that begin with an exclamation point. Text variables must have sufficient characters to populate the string and may not contain the comma delimiter.

/QUIET
This is useful for simulation scripts where console output is not desired. Typically, the /out or /symbol would be used to capture the data from the process variable.

/DEBUG
This will cause the output of tag information and other information as the data is acquired from the process variable. It can be helpful when determining if the VMIC address or other information is valid.

Examples:

```
$putit/quiet "CURRENT_SDR.INGOTID,ALLOY" "1234567890,LL"

$putit "CURRENT_SDR.INGOTID,ALLOY” @sys$input:
! Ingot ID
1234567890
! Alloy
LL


$MyTag= "CURRENT_SDR.INGOTID,ALLOY"
$putit 'MYTAG' @alloy.dat

$MYDATA = "1234567890,LL"
$putit/quiet "CURRENT_SDR.INGOTID,ALLOY" %MYDATA
```

The above examples write INGOTID and ALLOY elements into CURRENT_SDR process variable. The data is either from a file, a DCL symbol, or the command line.

5.1.3 GETIT

The GETIT utility is responsible for reading the value to any process variable. This includes a simple type (real, int, byte), or to an array of simple types, and finally, to any element or elements of a structure.

$GETIT[/DEBUG/SYMBOL=symbolname/OUT=Filename/QUIET] Tags

Tags
The Tags is the process variable desired to be acquired. If the process variable is an array, then the element index can be specified using square brackets: "[element number]". The element number always starts with one. If the process variable is an element, then the tag name followed by the desired elements of the structure can be specified, including elements of arrays contained within the structure. Tags can also be a file name that contains a list of elements from the process variable. If more than a single element of a process variable is to be returned, the string must be enclosed in quotes unless the tag string is passed via a data file.

/QUIET
This is useful for simulation scripts where console output is not desired. Typically, the /out or /symbol would be used to capture the data from the process variable.

/DEBUG
This will cause the output of tag information and other information as the data is acquired from the process variable. It can be helpful when determining if the VMIC address or other information is valid.

/SYMBOL=symbol name
This will write the data to a DCL local symbol with commas between the variables.

Examples:

```
$getit/quiet/symbol=GD "CURRENT_SDR.INGOTID,ALLOY"

$getit/symbol=GD @sys$input:
CURRENT_SDR.INGOTID,
ALLOY

$MyTag= "CURRENT_SDR.INGOTID,ALLOY"
$getit/symbol=GD 'MYTAG'

$getit/quiet/out=GD.TXT "CURRENT_SDR.INGOTID,ALLOY"
```

The above examples get INGOTID and ALLOY elements from CURRENT_SDR process variable and write them to the local DCL symbol "GD" or to a file called "GD.TXT". The GD.TXT file can be used by the PUTIT utility described earlier.

```
$ pass=1
$getit/symbol=PD -
```

```
    "CURRENT_SDR.PEDGPOS[''Pass'],PHORTORQUE[''Pass']"
$Epos= f$element(0,",",PD)
$Torque= f$element(1,",",PD)
$write sys$output "EdgPos: ''Epos'"
$write sys$output "Torque: ''Torque'"
```

The above example shows the usage with arrays. Note also the substitution of the DCL symbol "pass" into the string and the parsing of the symbol to get the particular values.

```
$if (f$search("sdr.tmp") .nes. "") then Delete sdr.tmp;*
$ GETIT/quiet/out=sdr.tmp  L2_SDR1
$ PUTIT/quiet CURRENT_SDR @sdr.tmp
```

The above example moves the L2_SDR1 process variable record structure to the CURRENT_SDR record structure.

5.1.4 RFM_PEEK UTILITY

The RFM_PEEK utility allows the caller to read anywhere in reflective memory and format the data out to the user. By default, if no format information is provided, the output is simply displayed in hex longwords.

```
$ RFM_PEEK
        /Start= reflective memory address
        [/Adapter=]
        [/Delimiter=]
        [/DATAfile=filename]
        [/DATASYM=DCL SYMBOL]
        [/FMT="format string"]
        [/FMTSYM= DCL symbol]
        [/COUNT= Byte count of read]
        [/GFLOAT]
```

/ADDRESS= Reflective memory address in hex

/ADAPTER= 5565 device, RBA0:, RBB0:, etc.. If not specified, RBA0: is used.

/COUNT= Optional count. If present, will limit the read regardless of how much data is read. If format string specifies more data than the count, the count takes precedence.

/DELIMITER= Character used as a delimiter to separate output data. Default is a comma.

/DATAFILE= Filename of the file to receive data read from memory. Default is sys$output.

/DATSYM= DCL Symbol to write formatted data to.

/FMT or /FMTSYM= Contains the format string used to format the data from the reflective memory. The format specifiers are as follows (similar to FORTRAN FORMAT Specifiers) and are comma separated. Repeat specifiers are supported for a single field, but no parenthetical repeat is supported. If no format information is provided, Z8.8 (hex) is used.

Zw= Integer*4 (hex)
Iw= Integer*4 (decimal)
Ww= Integer*2 (decimal)
Bw= Byte (decimal)
Fw.d= Real*4 (decimal IEEE float)
Xw= Skip over "w" bytes from read buffer
Cc= Character
D= 23 character OpenVMS Time Stamp

Where:
w= Width of field
d= Precision

/GFLOAT Floating point data should be converted from G_float

Examples:

$peek == "$ipd_prod:rfm_peek"

```
$ peek/adapter=rba0:/count=100/address=%x1000
$ peek/adapter=rba0:/count=100/address=%x1000/fmt="10z8"/count=80
$ peek/adapter=rba0:/count=100/address=%x1000/fmt="10I8"/count=80
$ peek/adapter=rba0:/count=100/address=%x1000/fmt="10I8"/count=80/datafile=test.log
$ peek/adapter=rba0:/address=%x1000/fmt="10z8"
$ peek/adapter=rba0:/address=%x1000/fmt="10f10.7/gfloat"
$ peek/adapter=rba0:/address=%x1000/fmt="10f10.7"
$ peek/address=%x1000/fmt="f10.7/count=4/datsym=speed
$ peek/address=%x1000/fmt="3f10.7,4I6"/count=28/datsym=speed
```

5.2 RFM_POKE UTILITY

The RFM_POKE utility allows the caller to write data in to reflective memory. If the memory location is not configured as a writeable region by the RFM utility, then the /FORCE switch must be specified to override this nodes write partition in reflective memory. **This utility should be used with caution.**

Syntax:

```
RFM_POKE
        /Start= reflective memory address
        [/Adapter=]
        [/Delimiter=]
        [/DATA="<delimited data>"]
        [/DATAfile=filename]
        [/DATASYM=DCL SYMBOL]
        [/FMT="format string"]
        [/FMTSYM= DCL symbol]
        [/COUNT= Byte count of read]
        [/FORCE]
```

/ADDRESS= Reflective memory address in hex

/ADAPTER= PCI-5565 device, RBA0:, RBB0:, etc.. If not specified, RBA0: is used.

/COUNT= Optional count. If present, will limit the write regardless of how much data is read. If format string specifies more data than the count, the count takes precedence.

/FORCE If present, will write to anywhere in reflective memory including memory range not allocated to this host (see RFM utility and driver partition table).

/DELIMITER=c Character used as a delimiter to parse data for the reflective memory. Default is a comma.

/DATAFILE=filename File containing the data to write to reflective memory.

/DATSYM= DCL Symbol to read formatted data from

/FMT or /FMTSYM= Contains the format string used to write the data for encoding into memory. The format specifiers are as follows (similar to FORTRAN FORMAT Specifiers) and are comma seperated. Repeat specifiers are supported for a single field, but no parenthetical repeat is supported. If no format information is provided, Z8.8 (hex) is used.

Zw= Integer*4 (hex)
Iw= Integer*4 (decimal)
Ww= Integer*2 (decimal)
Bw= Byte (decimal)
Fw.d= Real*4 (decimal IEEE float)
Xw= Skip over w bytes from read buffer
Cc= Character
D= 23 character OpenVMS Time Stamp

Where:

w= Width of field
d= Precision

Examples:

```
$ poke/adapter=rba0:/count=100/address=%x1000/debug/data=1234
$ poke/adapter=rba0:/count=100/address=%x1000/fmt="10z8"/count=80
$ poke/adapter=rba0:/count=100/address=%x1000/fmt="10I8"/count=80
$ poke/adapter=rba0:/count=100/address=%x1000-
   /fmt="10I8"/count=80/datafile=test.log
$ poke/adapter=rba0:/address=%x1000/fmt="10z8"
$ poke/adapter=rba0:/address=%x1000/fmt="10f10.7/gfloat"
$ poke/adapter=rba0:/address=%x1000/fmt="10f10.7"
$ poke/address=%x1000/fmt="f10.7/count=4/datsym=speed/debug
$ poke/adapter=rba0:/count=100/address=%x1000/data=<from time>
$ poke/force/address=%x1000/fmt="i10"/count=4/data=1023
```

5.3 RFM

The RFM utility provides a convenient DCL interface to control some of the capabilities of the PCI-5565 device. The utility makes use of the special I/O function codes to the driver.

5.3.1 RFM Command Syntax

The “< >” denote one of the options and are not part of syntax. The “[]” are optional parameters. One or more options may be specified on the command line. The adapter must be specified and may be a logical name or the actual PCI-5565 adapter device name (e.g., RBA0:, RBB0:, etc.).

$RFM adapter-
[/ENDIAN=<0,1>]-
[/MEMPAR="start1,size1,start2,size2,start3,size3"]-
[/NEWNODE]-
[/INTERRUPT]-
[/REFRESH]-
[/ROGUE=[1 or 2 D:1]-
[/DARK]-
[/STATUS]

Switch	Description
/ENDIAN=	Set little or big Endian mode. 0= little Endian, 1= big Endian
/MEMPAR= "start1,size1, start2,size2, start3,size3"	Set memory range partitions used the local host writeable regions within the reflective memory.
/NEWNODE /NONEWMODE	Enable/disable processing of node interrupt messages for nodes that join the reflective memory ring.
/INTERRUPT /NOINTERRUPT	Enable/disable network interrupts
/REFRESH	Refresh memory owned by this node for other nodes
/ROGUE /NOROGUE =[1 or 2 D:1]	Enable/disable Rogue packet functioning by this node. Default is rogue master one.
/LOOPBACK /NOLOOPBACK	Enable/disable local card loopback
/DARK /NODARK	Enable/disable dark on dark fiber operation
/STATUS	Display information about device

5.4 PSTCFG

The PSTCFG process is provided to support the accessing of the reflective memory using symbolic tags instead of absolute memory addresses in the reflective memory. The actual binding of the tag name to its address in reflective memory is done at runtime by the RFMGetReference function. This is a batch type function and any process that uses the tag method must be restarted if its tags are moved in reflective memory.

The PSTCFG process reads a comma delimited Tag CSV file (normally maintained by Excel and exported as a CSV file) and populates an RMS keyed file which defines a Tag within the reflective memory (name, description, data type, length, and offset in the memory). It will also support bit type tags that are packed into a long word. A header file that contains the definition of the bit word which contains the particular bit is also created. A small example of a CSV file is shown at end of this section. The RMS file is accessed by RFMGetReference function.

5.4.1 Command Syntax

The syntax for this command is:
$pstcfg/wrtlo=xxxx/wrthi=xxxx[/bits] filename.csv

Filename.csv= Input CSV file exported from Excel or other spread sheet or utilitiy.
/wrtlo= where in vmic reflective memory can write low value
/wrthi= where in vmic reflective memory can write high value
/bits= Write the bit header files

The write locations mark the tag as being a tag written or read by this local host. This version of the software uses a partition table in the driver to protect errant writes to process variables owned by other systems.

5.4.2 CSV File Column Definition

The columns for the CSV file are shown in the table below.

Column	Values	Description
Record Type	ELEM STRUC	Describes either a structure definition or an element of a structure or simple tag. Note: structure datatypes may not be possible over all platforms.
Data Type	INT- 2 byte integer DINT- 4 byte intger SINT- 1 byte integer REAL- Float CHAR- Text string BIT- Bit string START- Start Structure definition END- End Structure definition	The type of data contained in this tag. The floating point data types are normally expected to be IEEE float. The number of characters in the CHAR is defined using the Array size column. The structure definitions should be first in the CSV file. All BIT tags should be contained in a packed process variable of type INT or DINT. The packed variable is

		acquired and the bit is added to acquire its value. The /bits switch will define the layout of the bits.
RFM Addr	0 to cardsize	Reflective memory address on the PCI-5565 adapter. Should always attempt to use long word offset (e.g., modulus of 4 bytes). If this is a bit definition, the address is in the form of: “Address:bit number” in hex
Array size		Number of characters in a character string or the number of elements contained in the tag. Note: using arrays may not be possible over all platforms.
Descriptio n	80 characters	Description of tag. Should not contain any commas or apostrophes.
Units	20 characters	Documentation about the units of the tag. Should not contain any commas or apostrophes.
Source	12 character	Documentation about the node that owns this tag (e.g., writes to it). Should not contain any commas or apostrophes.

5.4.3 Example CSV file

```
! Tag Name,! Data,Variable,Array,RFM ADD,Description,Units,Source
!,! Type,Type,Size,(hex),,,
! Define structures first,,,,,,,
!,,,,,,,
PDO,Struct,START,,,Start structure definition,,
Produced_Length,Struct,REAL,,,Produced length,M,
Produced_Gauge,Struct,REAL,,,Produced gauge,mm,
Coil_ID,Struct,Char,12,,Coil identification,,
Alloy,Elem,Char,10,,Coils alloy,,
PDO,Struct,END,,,End structure definition,,
!# PLC to CPU1,,,,,,,
PLC_CPU1_GWStart,Elem,INT,0,0x000100,Guard Word Start,0,PLC
S1_ModelGauge_RFM,Elem,INT,0,0x000102,Model Gauge Stand 1,100 = 1mm,PLC
S2_ModelGauge_RFM,Elem,INT,0,0x000104,Model Gauge Stand 2,100 = 1mm,PLC
S3_ModelGauge_RFM,Elem,INT,0,0x000106,Model Gauge Stand 3,100 = 1mm,PLC
S1_AntiForce_RFM,Elem,INT,0,0x000108,Model Anticipated Force Stand 1,10 = 1MT,PLC
S2_AntiForce_RFM,Elem,INT,0,0x00010A,Model Anticipated Force Stand 2,10 = 1MT,PLC
S3_AntiForce_RFM,Elem,INT,0,0x00010C,Model Anticipated Force Stand 3,10 = 1MT,PLC
S12_ModelTenRef_RFM,Elem,INT,0,0x00010E,Model 1-2 Tension Reference,10 =1kg/cm2,PLC
S23_ModelTenRef_RFM,Elem,INT,0,0x000110,Model 2-3 Tension Reference,10 =1kg/cm2,PLC
TR_ModelTenRef_RFM,Elem,INT,0,0x000112,Model Reel Tension Reference,10 =1kg/cm2,PLC
S1_Model_RBRef_RFM,Elem,INT,0,0x000114,Model Roll Bending Reference Stand 1,10=1kg/cm2,PLC
F1_Top_Coolant_pat_1,Elem,INT,0,0x000198,Stand #1 Top Coolant Bit Fdbk (45),0,PLC
F1_TOP_DS_1,Elem,BIT,0,0x000198:00,F1 TOP drive side 1,Logical,PLC
F1_TOP_OS_1,Elem,BIT,0,0x000198:01,F1 TOP operator side 1,Logical,PLC
F1_TOP_DS_2,Elem,BIT,0,0x000198:02,F1 TOP drive side 2,Logical,PLC
F1_TOP_OS_2,Elem,BIT,0,0x000198:03,F1 TOP operator side 2,Logical,PLC
```

```
F1_TOP_DS_3,Elem,BIT,0,0x000198:04,F1 TOP drive side 3,Logical,PLC
F1_TOP_OS_3,Elem,BIT,0,0x000198:05,F1 TOP operator side 3,Logical,PLC
F1_TOP_DS_4,Elem,BIT,0,0x000198:06,F1 TOP drive side 4,Logical,PLC
F1_TOP_OS_4,Elem,BIT,0,0x000198:07,F1 TOP operator side 4,Logical,PLC
F1_TOP_DS_5,Elem,BIT,0,0x000198:08,F1 TOP drive side 5,Logical,PLC
F1_TOP_OS_5,Elem,BIT,0,0x000198:09,F1 TOP operator side 5,Logical,PLC
F1_TOP_DS_6,Elem,BIT,0,0x000198:0A,F1 TOP drive side 6,Logical,PLC
F1_TOP_OS_6,Elem,BIT,0,0x000198:01,F1 TOP operator side 6,Logical,PLC
F1_TOP_DS_7,Elem,BIT,0,0x000198:0C,F1 TOP drive side 7,Logical,PLC
F1_TOP_OS_7,Elem,BIT,0,0x000198:0D,F1 TOP operator side 7,Logical,PLC
F1_TOP_DS_8,Elem,BIT,0,0x000198:0E,F1 TOP drive side 8,Logical,PLC
F1_TOP_OS_8,Elem,BIT,0,0x000198:0F,F1 TOP operator side 8,Logical,PLC
F1_Top_Coolant_pat_2,Elem,INT,0,0x00019A,Stand #1 Top Coolant Bit Fdbk (44),Bitword,PLC
F1_TOP_DS_9,Elem,BIT,0,0x00019A:00,F1 TOP drive side 9,Logical,PLC
F1_TOP_OS_9,Elem,BIT,0,0x00019A:01,F1 TOP operator side 9,Logical,PLC
F1_TOP_DS_10,Elem,BIT,0,0x00019A:02,F1 TOP drive side 10,Logical,PLC
F1_TOP_OS_10,Elem,BIT,0,0x00019A:03,F1 TOP operator side 10,Logical,PLC
F1_TOP_DS_11,Elem,BIT,0,0x00019A:04,F1 TOP drive side 11,Logical,PLC
F1_TOP_OS_11,Elem,BIT,0,0x00019A:05,F1 TOP operator side 11,Logical,PLC
F1_TOP_DS_12,Elem,BIT,0,0x00019A:06,F1 TOP drive side 12,Logical,PLC
F1_TOP_OS_12,Elem,BIT,0,0x00019A:07,F1 TOP operator side 12,Logical,PLC
F1_TOP_DS_13,Elem,BIT,0,0x00019A:08,F1 TOP drive side 13,Logical,PLC
F1_TOP_OS_13,Elem,BIT,0,0x00019A:09,F1 TOP operator side 13,Logical,PLC
F1_TOP_DS_14,Elem,BIT,0,0x00019A:0A,F1 TOP drive side 14,Logical,PLC
F1_TOP_OS_14,Elem,BIT,0,0x00019A:01,F1 TOP operator side 14,Logical,PLC
F1_TOP_DS_15,Elem,BIT,0,0x00019A:0C,F1 TOP drive side 15,Logical,PLC
F1_TOP_OS_15,Elem,BIT,0,0x00019A:0D,F1 TOP operator side 15,Logical,PLC
!# CPU1 to CPU2,,,,,,,
CPU1_CPU2_GWStart,Elem,UDINT,0,0x004100,Guard Word Start,0,CPU1
S1_Roll_Gap_RFM12,Elem,REAL,0,0x004104,Current Roll Gap Stand 1,mm,CPU1
```

```
S2_Roll_Gap_RFM12,Elem,REAL,0,0x004108,Current Roll Gap Stand 2,mm,CPU1
S3_Roll_Gap_RFM12,Elem,REAL,0,0x00410C,Current Roll Gap Stand 3,mm,CPU1
Alloy_Grade_Num_RFM12,Elem,UDINT,0,0x004110,Alloy Grade,0,CPU1
S1_Motor_RPM_Ref_RFM,Elem,REAL,0,0x004114,Motor RPM Reference Stand 1,RPM,CPU1
S2_Motor_RPM_Ref_RFM,Elem,REAL,0,0x004118,Motor RPM Reference Stand 2,RPM,CPU1
S3_Motor_RPM_Ref_RFM,Elem,REAL,0,0x00411C,Motor RPM Reference Stand 3,RPM,CPU1
S1_Strip_Speed_RFM12,Elem,REAL,0,0x004120,Strip Speed Stand 1,MPM,CPU1
S2_Strip_Speed_RFM12,Elem,REAL,0,0x004124,Strip Speed Stand 2,MPM,CPU1
S3_Strip_Speed_RFM12,Elem,REAL,0,0x004128,Strip Speed Stand 3,MPM,CPU1
S1_Vern_Speed_RFM12,Elem,REAL,0,0x00412C,Stand 1 Operator Speed Vernier Change,Percent,CPU1
S2_Vern_Speed_RFM12,Elem,REAL,0,0x004130,Stand 2 Operator Speed Vernier Change,Percent,CPU1
S3_Vern_Speed_RFM12,Elem,REAL,0,0x004134,Stand 3 Operator Speed Vernier Change,Percent,CPU1
!CPU2,,,,,,,
CPU2_PLC_GWStart,Elem,UINT,0,0x004478,Start Guard,0,CPU2
S1_Cal_Scw_Pos_RFM,Elem,DINT,0,0x00447C,S1 Position (Actual & Calibrated),100 = 1 mm,CPU2
S1_UnCal_Scw_Pos_RFM,Elem,DINT,0,0x004480,S1 Position (Actual & Uncalibrated),100 = 1 mm,CPU2
S1_Scw_SetPoint_RFM,Elem,DINT,0,0x004484,S1 Position (Setpoint),100 = 1 mm,CPU2
S1_Scw_RCMinPos_RFM,Elem,DINT,0,0x004488,S1 Roll Change Minimum Position,100 = 1 mm,CPU2
CPU2_Alive_RFM,Elem,INT,0,0x00448C,CPU2 Heartbeat for the screw control in PLC,N/A,CPU2
S1_Scw_Jog_Ref_RFM,Elem,INT,0,0x00448E,S1 Current Jog Reference (Setpoint --- 100ct / m),cnts/m,CPU2
!CPU2,,,,,,,
S1_Screw_Packed_Bits1,Elem,UINT,0,0x004494,CPU2_to_PLC_Word_1,BitWord,CPU2
S1_Screw_Not_Ready,Elem,BIT,0,0x004494:01,S1 Screw Not Ready Or Faulted,Logical,CPU2
S1_Screw_Ready,Elem,BIT,0,0x004494:02,S1 Screw Ready and Enabled,Logical,CPU2
S1_Screw_In_Zone,Elem,BIT,0,0x004494:03,S1 Screw In Zone (Within Dead Band),Logical,CPU2
S1_Screw_In_RollChg_Pos,Elem,BIT,0,0x004494:05,S1 Screws In Position For Roll Change,Logical,CPU2
S1_Screw_At_Zero_Speed,Elem,BIT,0,0x004494:06,S1 Screw Zero Speed (Screw not Moving),Logical,CPU2
S1_Screw_Position_Move_CMD,Elem,BIT,0,0x004494:07,S1 Screw Position Move Command,Logical,CPU2
S1_Screw_Up_Limit_Stop,Elem,BIT,0,0x004494:08,S1 Screw Up Limit Stop,Logical,CPU2
S1_Screw_Down_Limit_Stop,Elem,BIT,0,0x004494:09,S1 Screw Down Limit Stop,Logical,CPU2
```

```
CPU2_PLC_GWEnd,Elem,INT,0,0x0044D2,Guard Word End,0,CPU2
TRACE_SAMPLES,Elem,DINT,0,0x0082D4,Gauge Trace Buffer sample counter,samples,CPU2
TRACE_BUF_COUNTER,Elem,UDINT,0,0x0082D8,New Buffer Counter Increments when coil complete,0,CPU2
TRACE_DISP,Elem,DINT,1500,0x0082DC,Sample displacement from head  end,Meters*10,CPU2
TRACE_MIN,Elem,DINT,1500,0x009A4C,Maximum deviation in sample,mm*1000,CPU2
TRACE_MAX,Elem,DINT,1500,0x00B1BC,Minimum deviation in sample,mm*1000,CPU2
TRACE_SIGMA,Elem,DINT,1500,0x00C92C,Standard deviation of sample,mm*1000,CPU2
TRACE_AVG,Elem,DINT,1500,0x00E09C,Average deviation in sample,mm*1000,CPU2
TRACE_LENGTH,Elem,DINT,0,0x00F80C,Incremental Length Measurement,M*10,CPU2
!# L2 to CPU1,,,,,,,
L2_SETUP_GDW1,Elem,DINT,0,0x014000,Setup guard word,0,L2
L2_S1_ModelGauge_RFM,Elem,DINT,0,0x014004,Model Gauge Stand 1,100 = 1mm,L2
L2_S2_ModelGauge_RFM,Elem,DINT,0,0x014008,Model Gauge Stand 2,100 = 1mm,L2
L2_S3_ModelGauge_RFM,Elem,DINT,0,0x01400C,Model Gauge Stand 3,100 = 1mm,L2
L2_S12_ModelTenRef_RFM,Elem,DINT,0,0x01401C,Model 1-2 Tension Reference,10 =1kg/cm2,L2
L2_S23_ModelTenRef_RFM,Elem,DINT,0,0x014020,Model 2-3 Tension Reference,10 =1kg/cm2,L2
L2_TR_ModelTenRef_RFM,Elem,DINT,0,0x014024,Model Reel Tension Reference,10 =1kg/cm2,L2
L2_S1_Model_RBRef_RFM,Elem,DINT,0,0x014028,Model Roll Bending Reference Stand 1,10=1kg/cm2,L2
L2_S2_Model_RBRef_RFM,Elem,DINT,0,0x01402C,Model Roll Bending Reference Stand 2,10=1kg/cm2,L2
L2_S3_Model_RBRef_RFM,Elem,DINT,0,0x014030,Model Roll Bending Reference Stand 3,10=1kg/cm2,L2
L2_T_Bar_Gauge_RFM,Elem,DINT,0,0x014040,Hot Entry Gauge,100=1mm,L2
L2_Strip_Width_RFM,Elem,DINT,0,0x014044,Hot Strip Width,mm,L2
L2_Entry_Temp_RFM,Elem,DINT,0,0x014048,Entry Temperature,DegC,L2
L2_Ingot_ID_RFM,Elem,DINT,2,0x01404C,Ingot ID,0,L2
L2_Alloy_Grade_RFM,Elem,DINT,0,0x014054,Alloy Grade,0,L2
L2_SETUP_ENTRY_LENGTH,Elem,DINT,0,0x014064,Entry length,mm,L2
L2_SETUP_ENTRY_TEMP,Elem,DINT,0,0x014068,Entry temperature,C,L2
L2_SETUP_ENTRY_CROWN,Elem,DINT,0,0x01406C,Entry Crown,mm*100,L2
L2_SETUP_PROD_GA,Elem,DINT,0,0x014070,Cold Product Gauge,mm*100,L2
L2_SETUP_PROD_WID,Elem,DINT,0,0x014074,Cold Product Width,mm,L2
```

```
L2_SETUP_TRIMMER_WID,Elem,DINT,0,0x014078,Side Trimmer Width Reference (Hot prod width),mm,L2
L2_SETUP_SHAPE_BIAS,Elem,DINT,0,0x01407C,Shape Bias,%,L2
L2_SETUP_GDW2,Elem,DINT,0,0x0140F8,Setup guard word end,0,L2
L2_HEARTBEAT,Elem,DINT,0,0x0140FC,Alpha Heartbeat,0,L2
TPAS_guard_word1,Elem,DINT,0,0x014290,Guard word #1 incremented for new setup,0,L2
TPAS_coil_id,Elem,Char,12,0x014294,Coil ID for this setup message,0,L2
TPAS_coil_setup_time,Elem,Char,16,0x014298,Coil setup time stamp from ATA level 2,0,L2
TPAS_alloy_index,Elem,REAL,0,0x0142A4,alloy index used in setting up gage,0,L2
TPAS_alloy,Elem,Char,8,0x0142B4,alloy for this schedule,0,L2
TPAS_transfer_gauge,Elem,REAL,0,0x0142B8,final gauge from roughing mill final pass,mm,L2
TPAS_exit_gauge,Elem,REAL,0,0x0142C0,scheduled exit gauge for finishing mill,mm,L2
TPAS_exit_gauge_ll,Elem,REAL,0,0x0142C4,exit gauge lower limit,mm,L2
TPAS_exit_gauge_ul,Elem,REAL,0,0x0142C8,exit gauge upper limit,mm,L2
TPAS_exit_gauge_blip_lim,Elem,REAL,0,0x0142CC,exit gauge blip count limit,0,L2
TPAS_transfer_width,Elem,REAL,0,0x0142D0,width from roughing mill final pass,mm,L2
L2_CUR_PDO,Record,PDO,0,,Current PDO record,,L2
L2_HIST_PDO,Record,PDO,6,,Last 6 produced PDO records,,L2
```

5.4.4 Example Command

The following is an example command file to regenerate the reflective memory map using a CSV file.

```
$! File: ReDo_All.com
$!
$! Abstract:
$! This command file creates a clean slate
$! from the CSV files.
$!
$! First it deletes the MCF and structures files
$! (RFMMCF.DAT and RFM_STRUCTURES.DAT)
$!
$! Second creates new ones
$!
$! Third run pstcfg for each file
$!
$! Files:
$!
$! LEVEL2.CSV
$!
$ set noon
$ ff[0,8]=12
$!
$ if(f$search("ipd_prod:RFM_STRUCTURES.DAT") .nes. "")
$ then
$  delete ipd_prod:rfm_structures.dat;*
$ endif
$!
$ if p1 .nes. ""
$ then
$   write sys$output "Deleteing old Database files"
$   Purge  ipd_prod:RFM_STRUCTURES.DAT
$   Purge  ipd_prod:RFMMCF.DAT
$   rename ipd_prod:RFM_STRUCTURES.DAT;0
ipd_prod:RFM_STRUCTURES.save
$   rename ipd_prod:RFMMCF.DAT;0 ipd_prod:RFMMCF.save
$ endif
$!
$! Delete old versions if present
$!
$ if(f$search("ipd_prod:RFM_STRUCTURES.DAT") .eqs. "")then -
      delete ipd_prod:RFM_STRUCTURES.DAT;*
$ if(f$search("ipd_prod:RFMMCF.DAT") .eqs. "")then -
      delete ipd_prod:RFMMCF.DAT;*
$!
$ write sys$output "Creating new Database files"
$ create/fdl=ipd_data:RFMMCF.FDL
$ create/fdl=ipd_data:RFM_STRUCTURES.FDL
$!
$ where = f$environment("Default")
$ temp= where - "]"
$ temp= temp + ".temp]"
$ show sym temp
$ if (f$trnlnm("rfm_temp") .eqs. "")
$ then
$   define/job rfm_temp 'temp'
$ endif
$ write sys$output "Deleting old include files in temp directory"
$ if(f$search("rfm_temp:*.inc") .nes."") then delete rfm_temp:*.inc;*
$!
$ pstcfg == "$''where'pstcfg"
$!
$!
$! RFM board
$!
$ write sys$output " "
$ write sys$output "RFM.csv"
$ write sys$output " "
$ pstcfg/wrtlo=%x00014000/wrthi=%x017fff/bits rfm.csv
```

5.4.5 Example Bit Definition File

Currently only FORTRAN type include files are supported. Contact IPACT if C header files are desired. Normally, the user would get the packed bit word and select the bit from the packed register using one of the library routines or Boolean arithmetic.

```
!
! Packed Bit word definition file
! (RFM_TEMP:F1_TOP_COOLANT_PAT_1_Bits.inc)
! Generated by PSTCFG
! Packed register:          F1_TOP_COOLANT_PAT_1
!   from:     198 to     199
!

! Discrete tag:                 F1_TOP_OS_8 (0x199 bit: 0d7)
! F1 TOP operator side 8
      Integer*4 F1_TOP_OS_8_Pos
      Parameter(F1_TOP_OS_8_Pos = 15)
      Integer*2 F1_TOP_OS_8_Msk
      Parameter(F1_TOP_OS_8_Msk = '8000'X)

! Discrete tag:                 F1_TOP_DS_8 (0x199 bit: 0d6)
! F1 TOP drive side 8
      Integer*4 F1_TOP_DS_8_Pos
      Parameter(F1_TOP_DS_8_Pos = 14)
      Integer*2 F1_TOP_DS_8_Msk
      Parameter(F1_TOP_DS_8_Msk = '4000'X)

! Discrete tag:                 F1_TOP_OS_7 (0x199 bit: 0d5)
! F1 TOP operator side 7
      Integer*4 F1_TOP_OS_7_Pos
      Parameter(F1_TOP_OS_7_Pos = 13)
      Integer*2 F1_TOP_OS_7_Msk
      Parameter(F1_TOP_OS_7_Msk = '2000'X)

! Discrete tag:                 F1_TOP_DS_7 (0x199 bit: 0d4)
! F1 TOP drive side 7
```

```
        Integer*4 F1_TOP_DS_7_Pos
        Parameter(F1_TOP_DS_7_Pos = 12)
        Integer*2 F1_TOP_DS_7_Msk
        Parameter(F1_TOP_DS_7_Msk = '1000'X)

! Discrete tag:                 F1_TOP_DS_6 (0x199 bit: 0d2)
! F1 TOP drive side 6
        Integer*4 F1_TOP_DS_6_Pos
        Parameter(F1_TOP_DS_6_Pos = 10)
        Integer*2 F1_TOP_DS_6_Msk
        Parameter(F1_TOP_DS_6_Msk = '0400'X)

! Discrete tag:                 F1_TOP_OS_5 (0x199 bit: 0d1)
! F1 TOP operator side 5
        Integer*4 F1_TOP_OS_5_Pos
        Parameter(F1_TOP_OS_5_Pos = 9)
        Integer*2 F1_TOP_OS_5_Msk
        Parameter(F1_TOP_OS_5_Msk = '0200'X)

! Discrete tag:                 F1_TOP_DS_5 (0x199 bit: 0d0)
! F1 TOP drive side 5
        Integer*4 F1_TOP_DS_5_Pos
        Parameter(F1_TOP_DS_5_Pos = 8)
        Integer*2 F1_TOP_DS_5_Msk
        Parameter(F1_TOP_DS_5_Msk = '0100'X)

! Discrete tag:                 F1_TOP_OS_4 (0x198 bit: 0d7)
! F1 TOP operator side 4
        Integer*4 F1_TOP_OS_4_Pos
        Parameter(F1_TOP_OS_4_Pos = 7)
        Integer*2 F1_TOP_OS_4_Msk
        Parameter(F1_TOP_OS_4_Msk = '0080'X)

! Discrete tag:                 F1_TOP_DS_4 (0x198 bit: 0d6)
! F1 TOP drive side 4
        Integer*4 F1_TOP_DS_4_Pos
        Parameter(F1_TOP_DS_4_Pos = 6)
        Integer*2 F1_TOP_DS_4_Msk
        Parameter(F1_TOP_DS_4_Msk = '0040'X)

! Discrete tag:                 F1_TOP_OS_3 (0x198 bit: 0d5)
! F1 TOP operator side 3
        Integer*4 F1_TOP_OS_3_Pos
        Parameter(F1_TOP_OS_3_Pos = 5)
        Integer*2 F1_TOP_OS_3_Msk
        Parameter(F1_TOP_OS_3_Msk = '0020'X)

! Discrete tag:                 F1_TOP_DS_3 (0x198 bit: 0d4)
! F1 TOP drive side 3
        Integer*4 F1_TOP_DS_3_Pos
        Parameter(F1_TOP_DS_3_Pos = 4)
        Integer*2 F1_TOP_DS_3_Msk
        Parameter(F1_TOP_DS_3_Msk = '0010'X)

! Discrete tag:                 F1_TOP_OS_2 (0x198 bit: 0d3)
! F1 TOP operator side 2
        Integer*4 F1_TOP_OS_2_Pos
        Parameter(F1_TOP_OS_2_Pos = 3)
        Integer*2 F1_TOP_OS_2_Msk
        Parameter(F1_TOP_OS_2_Msk = '0008'X)

! Discrete tag:                 F1_TOP_DS_2 (0x198 bit: 0d2)
! F1 TOP drive side 2
        Integer*4 F1_TOP_DS_2_Pos
        Parameter(F1_TOP_DS_2_Pos = 2)
        Integer*2 F1_TOP_DS_2_Msk
        Parameter(F1_TOP_DS_2_Msk = '0004'X)

! Discrete tag:                 F1_TOP_OS_1 (0x198 bit: 0d1)
! F1 TOP operator side 1
        Integer*4 F1_TOP_OS_1_Pos
        Parameter(F1_TOP_OS_1_Pos = 1)
        Integer*2 F1_TOP_OS_1_Msk
        Parameter(F1_TOP_OS_1_Msk = '0002'X)

! Discrete tag:                 F1_TOP_DS_1 (0x198 bit: 0d0)
! F1 TOP drive side 1
        Integer*4 F1_TOP_DS_1_Pos
        Parameter(F1_TOP_DS_1_Pos = 0)
        Integer*2 F1_TOP_DS_1_Msk
        Parameter(F1_TOP_DS_1_Msk = '0001'X)
```

5.5 RTREND Plotting Program for RFM Tags

RTREND is a real-time graphics program that plots up to 9 different RFM tags on the CRT screen using Regis graphics. It was developed as an aid to the software and modeling engineer.

All tags updated every second, which means that 11 minutes worth of data are plotted on the screen. At the end of the 11th minute, the graph shifts to the left by 5 minutes (the oldest 6 minutes of data disappear), and plotting continues for another 5 minutes.

Both integer and real tags are recognized and are converted correctly.

5.5.1 Usage

Limitations:

1. Recognizes only command line input. (Trend files not directly supported.)
2. Real-time plotting only. Plotting from historical data files is not available.

Usage:

RTREND /PHI=(plot high list) /PLO=(plot low list) /SCALE=(scaling factor list) “tag list”

Example:

```
$ rtrend /phi=(1000,1000,1000) /plo=(0,0,0) /scale=(10,10,10) -
   "learning_s1_force_rfm,learning_s2_force_rfm,learning_s3_force_rfm"
```

where,

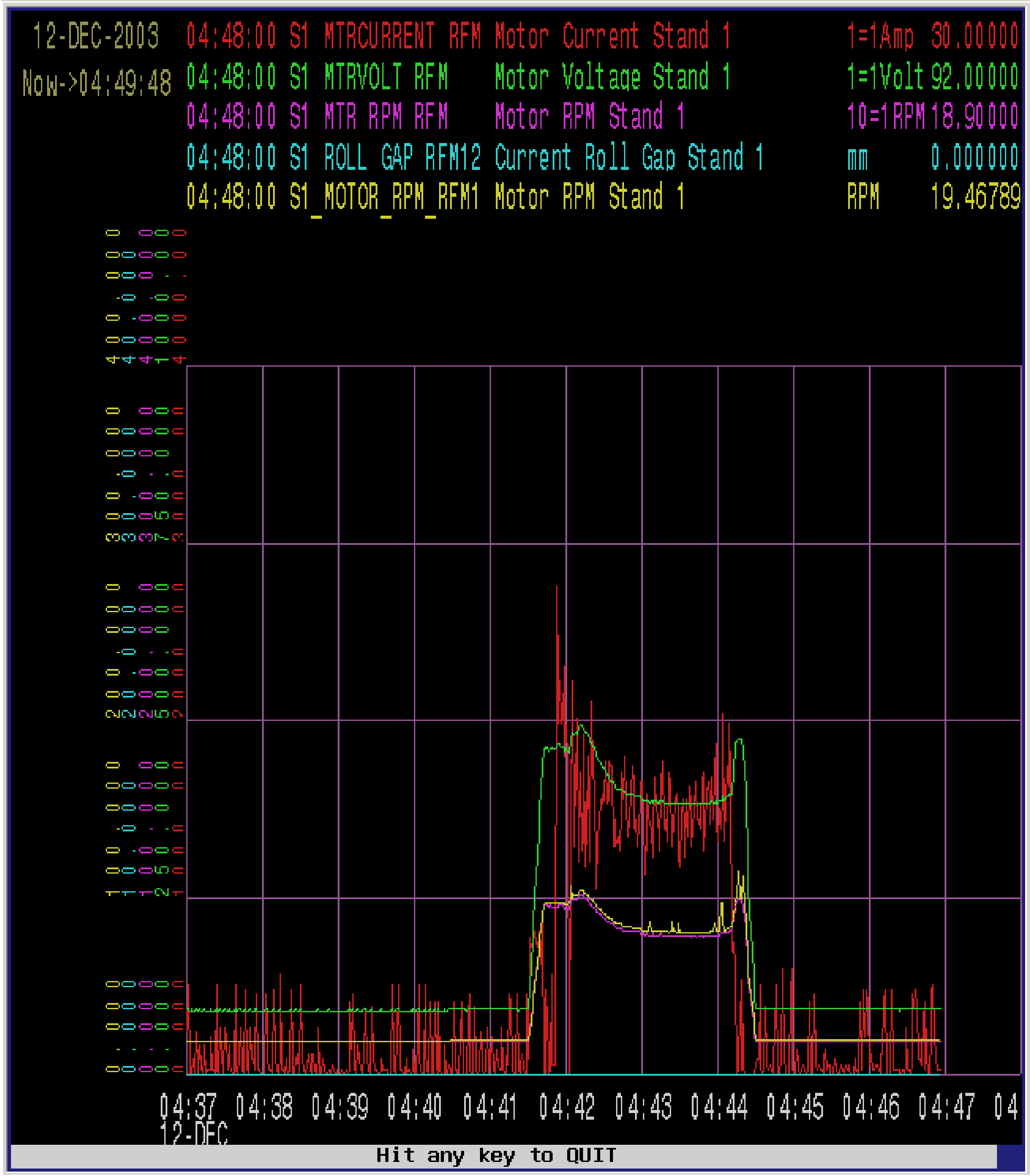
/PHI are the plot high limit values (parentheses *not required* if one tag). There must be the same number of plot high limit values as there are tags.

/PLO are the plot low limit values (parentheses *not required* if one tag)
If omitted, these values default to 0. If not omitted, there must be the same number of plot low limit values as there are tags.

/SCALE are the RFM scaling factors, tag values are ***divided*** by these factors (parentheses *not required* if one tag).
If omitted, these values default to 1. If not omitted, there must be the same number of scale values as there are tags.
Usage: Plotted Value = Tag Value / Scale Factor.

"…" is the list of 3 (maximum of 9) RFM tag names (quotes *not required* if one tag)

5.5.2 RTREND Example Plot



```
12-DEC-2003   04:48:00 S1 MTRCURRENT RFM Motor Current Stand 1        1=1Amp  30.00000
Now->04:49:48 04:48:00 S1 MTRVOLT RFM    Motor Voltage Stand 1        1=1Volt 92.00000
              04:48:00 S1 MTR RPM RFM    Motor RPM Stand 1            10=1RPM 18.90000
              04:48:00 S1 ROLL GAP RFM12 Current Roll Gap Stand 1     mm      0.000000
              04:48:00 S1_MOTOR_RPM_RFM1 Motor RPM Stand 1            RPM     19.46789
```

Figure 5.5-1 Sample RTREND Screen

6 Development Considerations

All reads and writes to reflective memory are done using longword moves (4 bytes). If a data item crosses over a longword boundary, the adjoining words must also be read and written resulting in two longwords being transmitted on the reflective memory ring. This may result in data being transmitted unexpectedly. The driver will perform the read, merge, and write for the data item that crosses the longword boundary.

Data that is written to reflective memory is written into a FIFO for transmission to the network. The actual transmission the fiber ring is asynchronous to the writing of the data by the host. Whenever a data item larger than 4 bytes or any data item that crosses a longword boundary it can possibly be transmitted in different data packets resulting in inconsistent data on the ring until subsequent packets are received to complete the update.

Users typically insert guard words into data items that span multiple longwords. The receiving nodes should only process the data when the guard words match.

Memory should be partitioned by node for each write area. No two nodes should have write access to the same memory space. Additionally, the memory should be contiguous by node. The driver attempts to ensure that no rogue process tries to write in a location not intended for write access by the host node. However, only three partitions are supported by the driver.

Some type of node synchronization should be designed when nodes join the reflective memory. The data from other nodes to the new joined node will not be valid until all other nodes write their memory locations and refresh the newly joined member's memory. A special driver function is available that will perform this (reads the current contents of memory and writes it back without modification).

6.1 Endian considerations

At the time of this writing, there are three versions of the PCI-5565 reflective memory adapters, PCI, VME, and Allen Bradley for Control Logix™. The Allen Bradley Control Logix version, CLB-5565, may or may not be readily available. The VME version, VME-5565, may be used in applications where the host processor may address memory differently (GE FANUC PAC90, GE 90/70, I/O Works INTEL based processor, or other embedded processor). Some experimentation may be needed to determine the best way to configure the byte ordering of the PCI reflective memory adapter. It may be necessary because of the different participants on the ring that the host applications may need to do byte swapping.

Index