PM7364

FREEDM-32

FREEDM SOFTWARE REFERENCE DESIGN

ISSUE 2

CONTENTS

1 INTRODUCTION

1.1 Scope

This document describes the routines and data structures which are necessary to interface a FREEDM to a PCI bus host. The routines are written and tested while running in a VxWorks real-time operating system (RTOS) environment. Source code for the software reference design is made available to customers using the FREEDM in their own designs. The source code can be used with any RTOS because the routines are written in C language and the software interfaces are designed to be independent of the RTOS.

The routines are tested on Pentium based motherboards with revision 2.1 compliant PCI local buses. The routines are intended to demonstrate the interface between the FREEDM and the host. The software reference design version 1.22 is described in this document, but the software for other devices on the reference design card and software work-arounds for any hardware functional deficiencies are not described.

1.2 Audience

The intended audience for this document is customers and engineers who want to make use of software routines, written and tested by PMC-Sierra, for interfacing the FREEDM to their system software.

1.3 Objective

The purpose of the FREEDM software reference design is to provide an example of the software interfaces for a typical application of the FREEDM. The source code is tested and debugged so that it can be readily used in any application that makes use of the FREEDM.

1.4 References

- PMC-960758, FREEDM-32 Data Sheet, October 1997, Issue 3
- PCI Special Interest Group, PCI BIOS Specification, August 26, 1994, Revision 2.1
- PMC-970281, FREEDM Programmer's Guide, February 1997, Issue 1

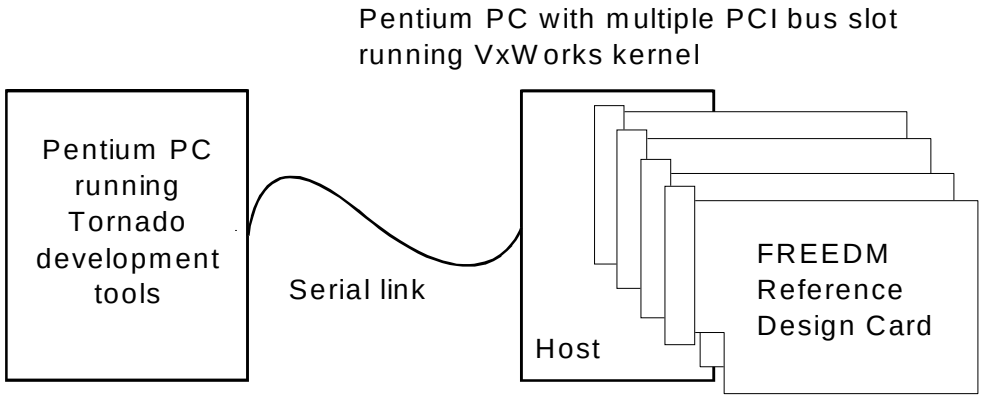
- PMC-970240, FREEDM-32 With TOCTL Reference Design, February 1997, Issue 1

2 DESIGN OVERVIEW

2.1 Reference Design Testbed

The reference design testbed in which the reference design software runs is shown in figure 1. The host is a Pentium based motherboard equipped with multiple PCI bus slots, each slot is capable of supporting a FREEDM reference design card. Since the interrupt pin of all PCI bus slots are multiplexed onto a single interrupt line of the host's Programmable Interrupt Controller, interrupt service routines of multiple reference design cards share the same interrupt line. Please refer to the hardware reference design document, PMC-970240, for details of the reference design card.

Figure 1. Simplified Block Diagram of Testbed



2.2 FREEDM Software Interfaces

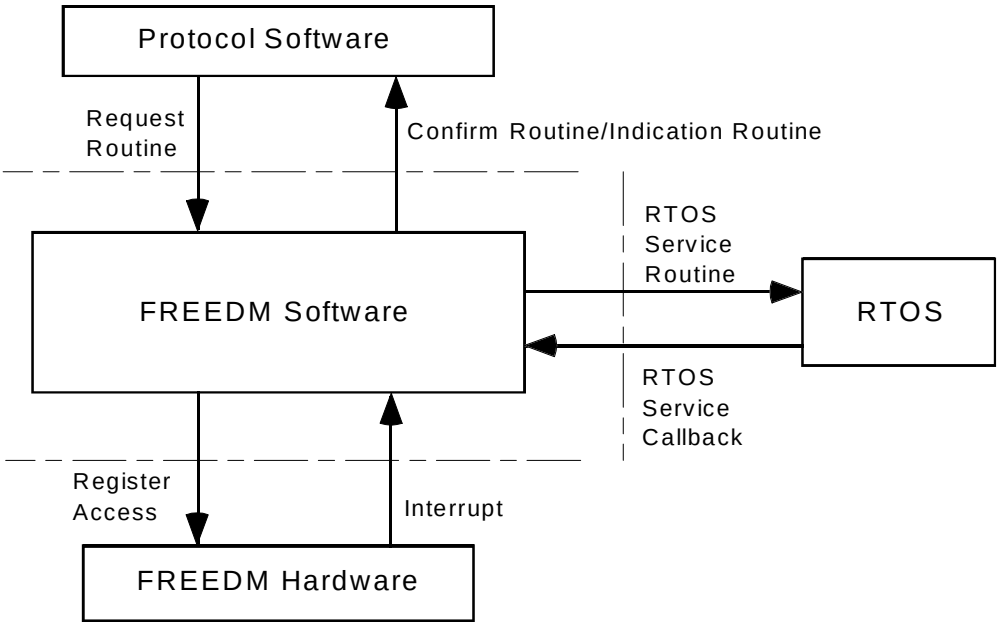
The FREEDM software reference design provides routines and data structures that are tightly coupled to the system software via the interfaces defined in this document. The routines do not run as a separate task within the RTOS but are callable by any task which makes use of them. For this reason they are designed to be re-entrant.

The following interfaces are defined (see figure 2):

- PCI hardware Interface
- RTOS services

- Protocol software Interface

Figure 2. FREEDM Software Interfaces



2.2.1 PCI Hardware Interface

The PCI hardware interface provides routines to read or write FREEDM registers within the PCI configuration space or the memory mapped address space. It also defines the template for an interrupt service routine which is called when an interrupt occurs on the PCI bus. The PCI BIOS functions are responsible for the operations in PCI configuration space, please refer to the PCI BIOS Specification for details of the functions.

2.2.2 RTOS Services Interface

The reference design software makes use of the following RTOS services:

- system timer
- memory allocation/deallocation
- translation between virtual and physical addresses
- installation and removal of an interrupt service routine

- deferred processing of interrupts

2.2.3 Protocol Software Interface

The protocol software interfaces to the FREEDM via request, confirm and indication primitives. These are defined in this document as the request, confirm and indication routines.

A request routine is simply a function called by the protocol software, whereby the FREEDM software carries out the request and returns a status. If the request is completed then the confirm primitive is included with return of the function call.

The confirm routines are necessary for the FREEDM software to return the status of a request to the protocol software when the status could not be provided during return from the request routine.

The indication routines are necessary for the FREEDM software to notify the protocol software of an event within the FREEDM hardware.

2.3 Overview of Software Features

Reset:

- Software reset of the FREEDM and the software.

Initialization:

- Initializes the FREEDM and software data structures (including queues and descriptor tables) that are associated with the FREEDM.

Activation/De-activation:

- Activation places the FREEDM in a state ready to transmit/receive packets across the PCI bus and enables the host to respond to interrupts generated by the FREEDM.

- De-activation disables the FREEDM and the interrupts.

Provisioning/Unprovisioning:

- Provision/unprovision FREEDM channels that are necessary to transmit and receive packets.

Transmit:

- Transmits packets by queuing packet references on ready queues. Free transmit buffers and descriptors on completion of data transfer across the PCI bus.

Receive:

- Receives packets by polling of packet references on ready queues or processing of receive interrupts. Replenish free receive buffers and descriptors after the host finishes reading the data from the buffers.

Interrupt Servicing:

- Clears the interrupt and passes the interrupt status to a deferred processing routine.

Interrupt Processing:

- Processing of the interrupt status within the deferred processing routine.

Diagnostics:

- Enables/disables internal diagnostic loopback on channel basis.
- Enables/disables external line loopback.
- Monitors the link activities.

Performance Monitor Statistics:

- Polls the counters to prevent them from overflow. The counts are then sent to the protocol software for accumulation.

For descriptions of the configurable features and operation of a FREEDM from a programmer's perspective, please refer to the FREEDM Programmer's Guide, PMC-970281.

2.4 Software State Diagram

Figure 3 shows the software state diagram. A state transition can occur only when a request routine successfully completes. The valid set of functions that can be called while the software is in each state is shown in table1.

Figure 3. FREEDM State Diagram

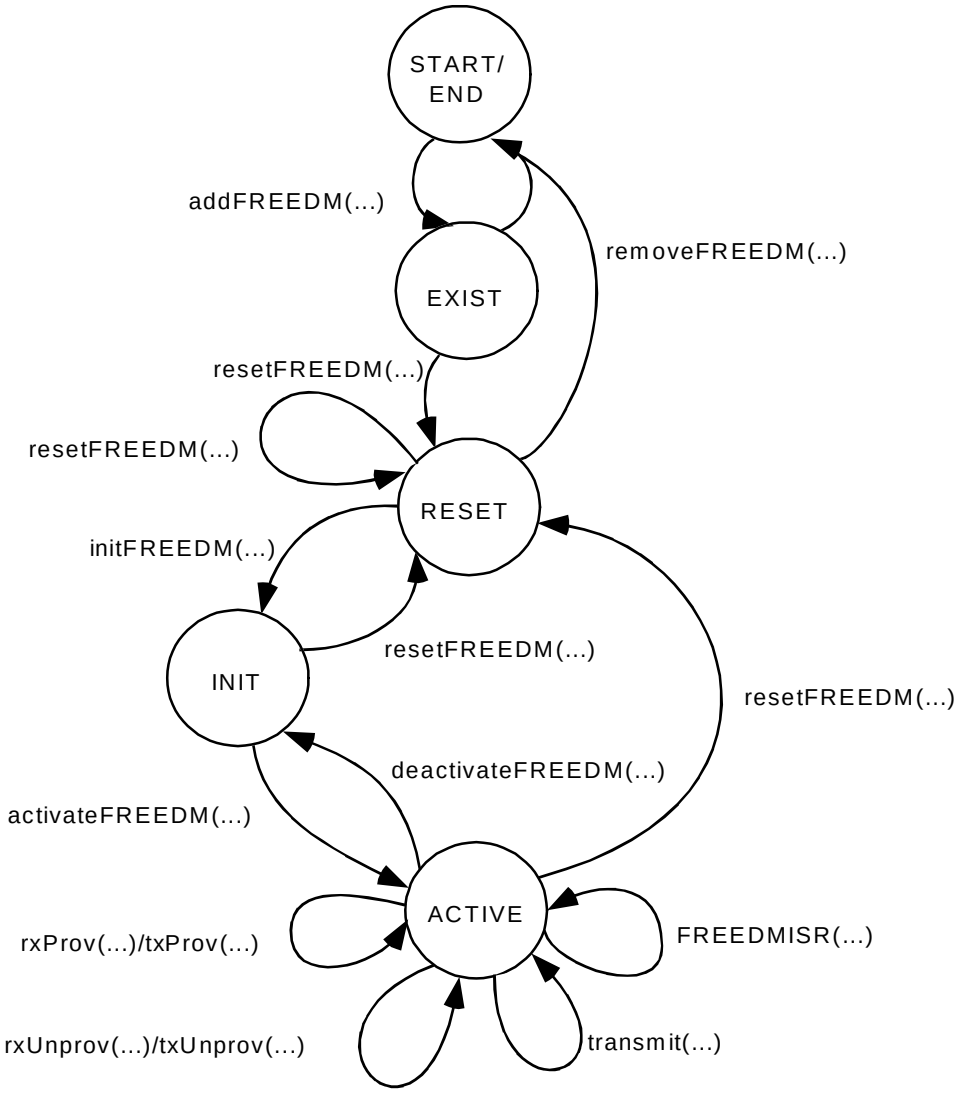


Table 1. Software State Definition

STATE	Description	Valid Functions
START/ END	The FREEDM hardware does not exist.	addFREEDM()
EXIST	The FREEDM hardware is found on the PCI bus.	resetFREEDM(), removeFREEDM()
RESET	The FREEDM hardware and the software is reset.	resetFREEDM(), initFREEDM(), removeFREEDM()
INIT	The data structures (descriptor tables, reference queues) are initialized.	activateFREEDM(), resetFREEDM()
ACTIVE	The FREEDM is activated and is ready to provision/unprovision channels, transmit/receive packets, process interrupts.	rxProv/txProv(), rxUnprov/txUnprov(), transmit(), FREEDMIsr(), deactivateFREEDM(), resetFREEDM()

3 PROTOCOL SOFTWARE INTERFACE

3.1 Data Structures and Constants

Software data structures are defined for code management and debugging purpose. They are optional. Customizations to the following data structures are necessary for different applications and systems specifications.

FREEDM Context Structure

The FREEDM context structure is allocated by the protocol software. Each structure contains the context of one FREEDM reference design card. The fields of the FREEDM context structure are defined as follows (data types and constants are included in Appendix A):

```
typedef struct{
   UBYTE                uState;
   DWORD                dInstance;
   DeviceHandle         dPCIHandle;
   DWORD                dTxDTSize;
   DWORD                dRxDTSize;
   DWORD                dRxSmallBufSize;
   DWORD                dRxLargeBufSize;
   TxPacketDescriptor   *pTxDTBase;
   DWORD                *pTxQBase;
   DWORD                *pTxAQBase;
   RxPacketDescriptor   *pRxDTBase;
   DWORD                *pRxQBase;
   DWORD                *pRxAQBase;
   TxPacket             *pTxPacketBase;
   DWORD                *pRxPacketBase;
   Block                *pTxBlkBase;
   Block                *pRxBlkBase;
   Block                *pTxHeadBlk[MAX_CHANNELS];
   Block                *pRxHeadBlk[MAX_CHANNELS];
   Block                *pFreeTxBlk;
   Block                *pFreeRxBlk;
   FREEDMQueue          TxAvailableQ;
   FREEDMQueue          TxFreeQ;
   FREEDMQueue          TxReadyQ;
   FREEDMQueue          RxAvailableQ;
   FREEDMQueue          RxLargeFreeQ;
   FREEDMQueue          RxSmallFreeQ;
   FREEDMQueue          RxReadyQ;
   TxChannelInfo        *pTxChannelInfo[MAX_CHANNELS];
   RxChannelInfo        *pRxChannelInfo[MAX_CHANNELS];
   RegInfo              Registers;
   DWORD                dClockMon;
   DWORD                dLinkMon;
   DWORD                dOFCounter;
   DWORD                dUFCounter;
```

```
    DWORD               dC1Counter;
    DWORD               dC2Counter;
    DWORD               dPMONStatus;
    DWORD               dCounterTimerID;
    DWORD               dActivityTimerID;
    DWORD               dProcessTimerID;
    BYTE                dCounterDelay;
    BYTE                dActivityDelay;
    BYTE                dProcessDelay;
} FREEDMContext;

/* Description of variables:                                          */
/* uState– software state                                             */
/* dInstance– device instance number                                  */
/* dPCIHandle– device handle number                                   */
/* dTxDTSize– size of the Tx descriptor table                         */
/* dRxDTSize– size of the Rx descriptor table                         */
/* dRxSmallBufSize– size of the Rx small data buffer                  */
/* dRxLargeBufSize– size of the Rx large data buffer                  */
/* *pTxDTBase– pointer to the base of Tx descriptor table             */
/* *pTxQBase- pointer to the base of Tx queue space                   */
/* *pTxAQBase- pointer to the base of Tx available queue              */
/* *pRxDTBase- pointer to the base of Rx descriptor table             */
/* *pRxQbase- pointer to the base of Rx queue space                   */
/* *pRxAQBase- pointer to the base of Rx available queue              */
/* *pTxPacketBase- pointer to the base of Tx packet ID table          */
/* *pRxPacketBase- pointer to the base of Rx packet ID table          */
/* *pTxBlkBase- pointer to the base of the Tx block structures        */
/* *pRxBlkBase- pointer to the base of the Rx block structures        */
/* *pTxHeadBlk[MAX_CHANNELS]- array of pointers to the first block*/
/*    of Tx partial packet buffer                                     */
/* *pRxHeadBlk[MAX_CHANNELS}- array of pointers to the first block*/
/*    of Rx partial packet buffer per channel                         */
/* *pFreeTxBlk- pointer to the next free Tx block                     */
/* *pFreeRxBlk- pointer to the next free Rx block                     */
/* TxAvailableQ- Tx available queue structure                         */
/* TxFreeQ- Tx descriptor free queue structure                        */
/* TxReadyQ- Tx descriptor ready queue structure                      */
/* RxAvailableQ- Rx available queue structure                         */
/* RxLargeFreeQ- Rx packet descriptor large buffer free queue         */
/*    structure                                                       */
/* RxSmallFreeQ- Rx packet descriptor small buffer free queue         */
/*    structure                                                       */
/* RxReadyQ- Rx packet descriptor ready queue structure               */
/* *pTxChannelInfo[MAX_CHANNELS]- array of pointers to Tx channel */
/*    provisioning information                                        */
/* *pRxChannelInfo[MAX_CHANNELs}- array of pointers to Rx channel */
/*    provisioning information                                        */
/* Registers- registers structure                                     */
/* dClockMon- register value of the clock activity monitor            */
/* dLinkMon- register value of the link activity monitor              */
/* dOFCounter- register value of PMON overflow counter                */
/* dUFCounter- register value of PMON underflow counter               */
/* dC1Counter- register value of PMON user configured counter 1   */
/* dC2Counter- register value of PMON user configured counter 2   */
/* dPMONStatus- register value of PMON status                         */
/* dCounterTimerID- ID of the PMON counters polling timer             */
```

```
/* dActivityTimerID- ID of the link activity polling timer        */
/* dProcessTimerID- ID of the queue processing timer              */
/* dCounterDelay- period of time (in ticks) to poll the PMON      */
/* dActivityDelay- period of time (in ticks) to poll the link     */
/*    activity monitor                                            */
/* dProcessDelay- period of time (in ticks) to process the queues */
```

The Tx and Rx available queues are software queue structures used as central pools of queue elements. They are implemented for queue element management.

The Tx and Rx packet ID tables are used to relate a packet ID to a descriptor reference. The packet ID of a Tx packet is assigned by the host and the ID of a Rx packet is assigned sequentially upon reception.

Provision Channel Info Structure

This structure contains information needed to provision a transmit/receive channel. It is allocated by the protocol software, a pointer to it is a parameter of the provision request. A pointer to the structure is dedicated for each channel and is valid until the removeFREEDM() call or the txUnprov()/rxUnprov() call. A total of 256 pointers (128 for transmit channels and 128 for receive channels) are contained within each FREEDM context structure. An invalid or undefined pointer assignment is NULL.

The fields of the transmit provision channel info are defined as follows:

```
typedef struct{
   UBYTE uTxChannel;
   UBYTE uLink;
   DWORD dTimeSlot;
   DWORD dTHDLChannelDataReg1Flag;
   DWORD dTHDLChannelDataReg2Flag;
   DWORD dTHDLChannelDataReg3;
   WORD  dNoOfBlocks;
   WORD  dUnderflow;
}TxChannelInfo;

/* Description of variables:                                      */
/* uTxChannel- Tx channel number to be provisioned                */
/* uLink- Tx link mapped to uTxChannel                            */
/* dTimeslot- Tx timeslots of uLink mapped to uTxChannel          */
/* dTHDLChannelDataReg1Flag- CRC[1:0], IDLE and DELIN of          */
/*    THDL Indirect Channel Data #1 register                      */
/* dTHDLChannelDataReg2Flag- 7BIT, PRIORITYB, INVERT and DFCS of  */
/*    THDL Indirect Channel Data #2 register                      */
/* dTHDLChannelDataReg3- value of THDL Indirect Channel Data #3   */
/*    register                                                    */
/* dNoofBlocks- length of the Tx partial packet buffer            */
/* dUnderflow- underflow status from the Tx descriptor reference  */
```

The fields of the receive provision channel info are defined as follows:

```
typedef struct{
   UBYTE uRxChannel;
   UBYTE uLink;
   DWORD dTimeSlot;
   UBYTE uCDLBEN;
   DWORD dRHDLChannelDataReg1Flag;
   DWORD dRHDLChannelDataReg2;
   WORD  dNoOfBlocks;
}RxChannelInfo;

/* Description of variables:                                        */
/* uRxChannel- Rx channel number to be provisioined                 */
/* uLink- Rx link mapped to uRxchannel                              */
/* dTimeSlot- Rx timeslots of uLink mapped to uRxChannel            */
/* uCDLBEN- diagnostic loopback enable bit                          */
/* dRHDLChannel1DataReg1Flag- CRC[1:0], STRIP and DELIN of          */
/*    RHDL Indirect Channel Data Register #1                        */
/* dRHDLChannelDataReg2- value of RHDL Indirect Channel Data        */
/*    Register #2                                                   */
/* dNoOfBlocks- length of the Rx partial packet buffer              */
```

For a channelised T1 link, time-slots 1 to 24 are valid. For a channelised E1 link, time-slots 1 to 31 are valid. For unchannelised links, only time-slot 0 is valid. Each time-slot corresponds to a bit in the dTimeSlot field. For example, when time-slot 0 is configured, the least significant bit in dTimeSlot is set to 1 .

FREEDM Queue Structure

The FREEDM queue structure is allocated by the protocol software but initialized during transition to the INIT state. It contains the virtual memory address of the queue base, the virtual memory addresses of the queue index registers, and the values of these registers. This queue structure is contained within the FREEDM context structure. The fields of the FREEDM queue are defined as follows:

```
typedef struct{
   WORD     dHeadroom;
   WORD     dBatchSize;
   BYTE     dReadCount;
   BYTE     dWriteCount;
   DWORD    dQSize;
   DWORD    *pQBase;
   DWORD    *pStartReg;
   DWORD    *pWriteReg;
   DWORD    *pReadReg;
   DWORD    *pEndReg;
   DWORD    dStart;
   DWORD    dWrite;
   DWORD    dRead;
   DWORD    dEnd;
```

```
}FREEDMQueue;

/* Description of variables:                                          */
/* dReadRoom- number of references to be read                         */
/* dWriteRoom- number of spaces to be written                         */
/* dBatchSize- for PCI bus performance, read and write index          */
/*    registers are written in batch                                  */
/* dReadCount- counter of reads before accessing registers            */
/* dWriteCount- counter of writes before accessing registers          */
/* dQSize- size of the queue                                          */
/* *pQBase- pointer to the queue base register                        */
/* *pStartReg- pointer to the start index register                    */
/* *pWriteReg- pointer to the write index register                    */
/* *pReadReg- pointer to the read index register                      */
/* *pEndReg- pointer to the end index register                        */
/* dStart- shadow of start index in the host                          */
/* dWrite- shadow of write index in the host                          */
/* dRead- shadow of read index in the host                            */
/* dEnd- shadow of end index in the host                              */
```

FREEDM Block Structure

The FREEDM block structure contains information necessary to identify the partial packet buffer blocks within the RDHL and TDHL of each channel. This structure is used as a software copy of the partial packet buffer blocks within the internal RAM of the FREEDM. There are 512 blocks for receive and 512 blocks for transmit direction. The head pointer for each partial packet buffer list is contained within the FREEDM context structure in fields: pTxHead[MAX_CHANNELS] and pRxHead[MAX_CHANNELS]. These pointers are NULL until they are assigned to form a partial packet buffer during channel provisioning. The fields of a FREEDM block are defined as follows:

```
typedef struct BlockStruct{
   DWORD dBlockNum;
   struct   BlockStruct *pNext;
}Block;

/* Description of variables:                                          */
/* dBlockNum- 1<=block number<=512                                    */
/* *pNext- pointer to the next block, if there is any                 */
```

FREEDM Descriptor Structure

The FREEDM descriptor structure is allocated by the protocol software. The descriptors are found in the descriptor tables. A pointer to this structure is used in transmit request/confirm and receive indication routine. The field of the descriptor structure is an array of the four DWORDS of a descriptor:

```
typedef struct  {
   DWORD dTDArray[4];
} TxPacketDescriptor;

/* Description of variable:                                        */
/* dTDArray[4]- an array of four DWORDS of a Tx descriptor         */

typedef struct  {
   DWORD dRDArray[4];
} RxPacketDescriptor;

/* Description of variable:                                        */
/* dRDArray[4]- an array of four DWORDS of a Rx descriptor         */
```

FREEDM Packet Structure

The FREEDM packet structure is allocated by the protocol software. A pointer to this structure is used in transmit request/confirm and receive indication routine. The fields of the packet structure are defined as follows:

```
typedef struct{
   DWORD    dPacketID;
   Buffer   *pHeadAddr;
   UBYTE    uTxChannel;
   BYTE     dStatus;
   WORD     dPriorityFlag;
   WORD     dABTIOCFlag;
}TxPacket;

/* Description of variables:                                       */
/* dPacketID- ID of a packet that transmitted by user              */
/* *pHeadAddr- pointer to the first buffer structure               */
/* uTxChannel- channel number that the packet to be transmitted to*/
/* dStatus- status to indicate result of Tx                        */
/* dPriorityFlag- set priority bit on Bit 10 of 2nd DWORD of Tx    */
/*    descriptor                                                   */
/* dABTIOCFlag- set ABT bit on Bit 15 and IOC bit on Bit 14 of 3rd */
/*    DWORD of Tx descriptor                                       */

typedef struct{
   DWORD    dPacketID;
   Buffer   *pHeadAddr;
   UBYTE    uRxChannel;
   BYTE     dStatus;
   BYTE     dOffset;
}RxPacket;

/* Decription of variables:                                        */
/* dPacketID- ID of a packet received                              */
/* *pHeadAddr- pointer to the first buffer structure               */
/* uRxChannel- channel number that the packet is received from     */
/* dOffset- byte offset of data packet from the start of buffer    */
/* dStatus- status of the Rx packet decriptor reference            */
```

The fields of the buffer structure are defined as follows:

```
typedef struct BufferStruct{
   WORD     dBufSize;
   BYTE     *pData;
   struct   BufferStruct   *pNext;
}Buffer;

/* Description of variables:                                          */
/* dBufSize- size of buffer                                           */
/* *pData- pointer to the buffer data location                        */
/* *pNext- pointer to the next buffer, if there is any                */
```

FREEDM Interrupt Context Structure

The FREEDM interrupt context structure is allocated during initialization when the routine installISR is called. It contains information needed for the interrupt process. A pointer to this structure is passed into the interrupt service routine and the deferred process routine as an argument. The fields of the interrupt context are defined as follows:

```
typedef struct{
   FREEDMContext  *pFreedm;
   DWORD          dIntStatus;
}IntContext;

/* Description of variables:                                          */
/* *pFreedm- pointer to the FREEDM context structure                  */
/* dIntStatus- value of the interrupt status register                 */
```

FREEDM Register Info Structure

The FREEDM register info structure is allocated by the protocol software. It contains registers values needed for FREEDM initialization. This structure is contained within the FREEDM context structure. The fields of the register info are defined as follows:

```
typedef struct{
   DWORD dIntEnable;
   DWORD dRCASLink[MAX_LINKS];
   DWORD dTCASLink[MAX_LINKS];
   DWORD dRCASFrameBitThres;
   DWORD dTCASFrameBitThres;
   DWORD dTCASIdleTSFillData;
   DWORD dRMACCtrl;
   DWORD dTMACCtrl;
   DWORD dGPICCtrl;
   DWORD dRHDLMaxLength;
   DWORD dRHDLConfig;
   DWORD dTHDLConfig;
   DWORD dPerfMonCtrl;
}RegInfo;
```

```
/* Description of variables:                                          */
/* dIntEnable- interrupts to be enabled                               */

/* dRCASLink[MAX_LINKS]- value of RCAS Link #n Configuraton           */
/*    register                                                        */
/* dTCASLink[MAX_LINKS]- value of TCAS Link #n Configuration          */
/*    register                                                        */
/* dRCASFrameBitThres- threshold used to detect framing bits/bytes*/
/* dTCASFrameBitThres- threshold used to detect framing bits/bytes*/
/* dTCASIdleTSFillData- data to be written to disabled time-slots */
/* dRMACCtrl- configuration of RMAC Control register                  */
/* dTMACCtrl- configuration of TMAC Control register                  */
/* dGPICCtrl- configuration of GPIC Control register                  */
/* dRHDLMaxLength- configuration of RHDL Maximum packet length    */
/* dRHDLConfig- value of RHDL Configuration register                  */
/* dTHDLConfig- value of THDL Config register                         */
/* dPerfMonCtrl- counters to be configured for PMON control           */
```

3.2 Request Routines

Variable that is global to all the request routines:

```
typedef enum {SUCCESS, PENDING, FAILURE}RC;
```

When FAILURE is returned, a global integer errno is set to propagate error information.

3.2.1 Add FREEDM

The function addFREEDM adds the FREEDM context to the software. The FREEDM changes from START/END state to EXIST state.

Function Prototype:

```
RC addFREEDM(
   FREEDMContext  *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be added.
```

Return Value:

```
RC (return code):
   SUCCESS or FAILURE if the FREEDM is not in START/END state or if the
routine addPCIDevice fails or if the installTimer fails .
```

Modified Value:

The PCI handle for the FREEDM and the state of the FREEDM.

Side Effects:

None.

Caveat:

The FREEDM must be physically plugged into the host to complete the addFREEDM function. The protocol software must have initialized the following fields within the FREEDM context before calling addFREEDM:

```
   UBYTE uState,
   BYTE  dInstance
```

Pseudo-Code:

```
RC addFREEDM(
   FREEDMContext  *pFreedm)
{
   pFreedm->dPCIHandle <- addPCIDevice(pFreedm->dInstance, DEVICE_ID,
      VENDOR_ID)
   If pFreedm->dPCIHandle is less than 0, return FAILURE
   Install pollCounter and queueProcess timers
   If installTimer fails, clean up and return FAILURE
   Assign timer ID within the FREEDM context
   Modify the state of FREEDM to EXIST
   Return SUCCESS
}
```

3.2.2 Remove FREEDM

The function removeFREEDM removes the FREEDM context from the software .

Function Prototype:

```
RC removeFREEDM(
   FREEDMContext *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be removed.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is in the INIT or ACTIVE state.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Caveat:

```
The FREEDM must be in the EXIST or the RESET state before this function
is run.
```

Pseudo-Code:

```
RC removeFREEDM(
   FREEDMContext *pFreedm)
{
   If FREEDM is in INIT or ACTIVE state, set errno and return FAILURE
   Remove the timers set for the FREEDM
   If fail to remove timers, return FAILURE
   Else return SUCCESS
}
```

3.2.3 Reset FREEDM

The function resetFREEDM is responsible for resetting the FREEDM under software control.

Function Prototype:

```
RC resetFREEDM(
   FREEDMContext *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be reset.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is in the START/END state
```

Modified Value:

```
The state of the FREEDM, from EXIST/INIT/ACTIVE to RESET.
```

Side Effects:

```
None.
```

Caveat:

```
The FREEDM must not be in the START/END state before this function is
run.
```

Pseudo-Code:

```
RC resetFREEDM(
   FREEDMContext *pFreedm)
{
   If FREEDM is in START/END state, set errno and return FAILURE
   Disable the interrupts
   RMAC control register.ENABLE bit <- 0
   TMAC Control register.ENABLE bit <- 0
   FREEDM Master Reset and Identity register.RESET bit <- 1
   FREEDM Master Reset register.RESET bit <- 0
   If FREEDM is in the INIT or ACTIVE state, deallocate the memory space
      that has been allocated during initialization
   If FREEDM is in the ACTIVE state, remove timers and ISR that have
      been installed during activation.
   Change the state of the FREEDM to RESET
   Return SUCCESS
}
```

3.2.4 Initialize FREEDM

The initialization routine is responsible for initialization of the FREEDM data structures.

Function Prototype:

```
RC initFREEDM(
   FREEDMContext *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be initialized.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is not in the RESET state.
```

Modified Value:

```
The state of the FREEDM, from RESET to INIT.
```

Side Effects:

```
None.
```

Caveat:

The protocol software must have initialized the following fields within the FREEDM context before calling initFREEDM:

```
FREEDMContext:
   DWORD dTxDTSize; /* at max 16384 */
   DWORD dRxDTSize; /* at max 16384 */

   FREEDMQueue TxAvailableQ:  DWORD dStart;
                              DWORD dQSize; /* equals to dTxDTSize */

   FREEDMQueue TxFreeQ:       DWORD ddStart;
                              DWORD dQSize;

   FREEDMQueue TxReadyQ:      DWORD dStart;
                              DWORD dQSize;

   FREEDMQueue RxAvailableQ:  DWORD dStart;
                              DWORD dQSize; /* equals to dRxDTSize */

   FREEDMQueue RxLargeFreeQ:  DWORD dStart;
                              DWORD dQSize;

   FREEDMQueue RxSmallFreeQ:  DWORD dStart;
                              DWORD dQSize;

   FREEDMQueue RxReadyQ:      DWORD dStart;
                              DWORD dQSize;

   RegInfo Registers:         DWORD dIntEnable;
                              DWORD dTCASLink[MAX_LINKS];
                              DWORD dRCASLink[MAX_LINKS];
                              DWORD dTCASFrameBitThres;
                              DWORD dTCASIdleTSFillData;
                              DWORD dRCASFrameBitThres;
                              DWORD dTMACCtrl;
                              DWORD dRMACCtrl;
                              DWORD dGPICCtrl;
                              DWORD dTHDLConfig;
                              DWORD dRHDLConfig;
                              DWORD dRHDLMaxLength;
                              DWORD dPerfMonCtrl;
```

Pseudo-Code:

```
RC initFREEDM(
   FREEDMContext *pFreedm)
{
   If the FREEDM is not in RESET state,  set errno and return FAILURE

   /* Check the user inputs */
   /* for both tx and rx direction - check max queue size*/
   queue limit <- pFreedm->pTxQBase + MAX_QUEUE_SPACE (256K)
   If any one of the Tx queues exceeds the queue limit, set
      errno and return FAILURE

   queue limit <- pFreedm->pRxQBase + MAX_QUEUE_SPACE (256K)
   If any one of the Rx queues exceeds the queue limit, set
      errno and return FAILURE

   /* for both tx and rx direction - check overlap */
   If start index of the queue overlap another queue space, set errno
      and return FAILURE

   /* check size of rx large and small free queue */
   If both rx large and small free queue size = ZERO, set errno and
      return FAILURE

   /* Initialization of base addresses */
   pTxDTBase <- allocate memory space for Tx descriptor table
   pTxQBase <- allocate memory space for Tx reference queues
   pTxAQBase <- allocate memory space for Tx available queue
   pRxDTBase <- allocate memory space for Rx descriptor table
   pRxQBase <- allocate memory space for Rx reference queues
   pRxAQBase <- allocate memory space for Rx available queue
   Allocate memory space for TxPacket table and rx packet ID table
   Allocate memory space for free block structure

   /* covert the virtual address to physical address if the virt addr
      is different from the phys addr (it is the same in VxWorks) */
   phyTxDTBase <- Convert address of *pTxDTBase to physical address
   TMAC Transmit Descriptor Table Base LSW register <- the least
   significant word of phyTxDTBase
   TMAC Transmit Desciptor Table Base MSW register <- the most
   significant word of phyTxDTBase

   /* convert the virtual addrss to physical address if the virt addr
      is different from the phys addr (it is the same in VxWorks) */
   phyTxQBase <- Convert address of *pTxQBase to physical address
   TMAC Transmit Queue Base LSW register <- the least significant
      word of phyTxQBase
   TMAC Transmit Queue Base MSW register <- the most significant
      word of phyTxQBase

   /* covert the virtual address to physical address if the virt addr
      is different from the phys addr (it is the same in VxWorks) */
   phyRxDTBase <- Convert address of *pRxDTBase to physical address
   RMAC Packet Descriptor Table Base LSW register <- the least
   significant word of phyRxDTBase
```

```
        RMAC Receive Desciptor Table Base MSW register <- the most
        significant word of phyRxDTBase

        /* covert the virtual address to physical address if the virt addr
           is different from the phys addr (it is the same in VxWorks) */
        phyRxQBase <- Convert address of *pRxQBase to physical address
        RMAC Receive Queue Base LSW register <- the least significant
           word of phyRxQBase
        RMAC Receive Queue Base MSW register <- the most significant
           word of phyRxQBase

        /* Rx Ready Queue Initialization */
        dRxReadyPhyQSize <- RxReadyQ.dQSize + 1
        RxReadyQ.dWrite <- RxReadyQ.dStart
        RxReadyQ.dRead <- RxReadyQ.dStart + RxReadyQ.dQSize
        RxReadyQ.dEnd <- RxReadyQ.dStart + dRxReadyPhyQSize
        RMAC PDRRQ Start <- RxReadyQ.dStart
        RMAC PDRRQ Write <- RxReadyQ.dWrite
        RMAC PDRRQ Read <- RxReadyQ.dRead
        RMAC PDRRQ End <- RxReadyQ.dEnd

        /* Rx Small Buffer Free Queue Initialization */
        dRxSmallPhyQSize <- RxSmallFreeQ.dQSize + 1
        RxSmallFreeQ.dWrite <- RxSmallFreeQ.dStart
        RxSmallFreeQ.dRead <- RxSmallFreeQ.dStart + RxSmallFreeQ.dQSize
        RxSmallFreeQ.dEnd <- RxSmallFreeQ.dStart + dRxSmallPhyQSize
        RMAC PDRSBFQ Start <- RxSmallFreeQ.dStart
        RMAC PDRSBFQ Write <- RxSmallFreeQ.dWrite
        RMAC PDRSBFQ Read <- RxSmallFreeQ.dRead
        RMAC PDRSBFQ End <-  RxSmallFreeQ.dEnd

        /* Put references on the Rx small buffer free queue */
        Use a FOR loop to put references on the Rx small buffer free queue

        /* Rx Large Buffer Free Queue Initialization */
        dRxLargePhyQSize <- RxLargeFreeQ.dQSize + 1
        RxLargeFreeQ.dWrite <- RxLargeFreeQ.dStart
        RxLargeFreeQ.dRead <- RxLargeFreeQ.dStart + RxLargeFreeQ.dQSize
        RxLargeFreeQ.dEnd <- RxLargeFreeQ.dStart + dRxLargePhyQSize
        RMAC PDRLBFQ Start <- RxLargeFreeQ.dStart
        RMAC PDRLBFQ Write <- RxLargeFreeQ.dWrite
        RMAC PDRLBFQ Read <- RxLargeFreeQ.dRead
        RMAC PDRLBFQ End <- RxLargeFreeQ.dEnd

      /* Put references on the Rx large buffer free queue */
        Use a FOR loop to put references on the Rx large buffer free queue

      /* Rx Available Queue Initialization */
        dRxAvailablePhyQSize <- RxAvailableQ.dQSize + 1
        RxAvailableQ.dWrite <- RxAvailableQ.dStart
        RxAvailableQ.dRead <- RxAvailableQ.dStart + RxAvailableQ.dQSize
        RxAvailableQ.dEnd <- RxAvailableQ.dStart + dTxAvailablePhyQSize

      /* Put references on the Rx available queue */
        Use a FOR loop to assign references on the Rx available queue

      /* Tx Ready Queue Initialization */
```

```
        dTxReadyPhyQSize <- TxReadyQ.dQSize + 1
        TxReadyQ.dWrite <- TxReadyQ.dStart
        TxReadyQ.dRead <- TxReadyQ.dStart + TxReadyQ.dQSize
        TxReadyQ.dEnd <- TxReadyQ.dStart + dTxReadyPhyQSize
        TMAC TDRRQ Start <- TxReadyQ.dStart
        TMAC TDRRQ Write <- TxReadyQ.dWrite
        TMAC TDRRQ Read <- TxReadyQ.dRead
        TMAC TDRRQ End <- TxReadyQ.dEnd

        /* Tx Free Queue Initialization */
        dTxFreePhyQSize <- TxFreeQ.dQSize + 1
        TxFreeQ.dWrite <- TxFreeQ.dStart
        TxFreeQ.dRead <- TxFreeQ.dStart + TxFreeQ.dQSize
        TxFreeQ.dEnd <- TxFreeQ.dStart + dTxFreePhyQSize
        TMAC TDRFQ Start <- TxFreeQ.StartÎndex
        TMAC TDRFQ Write <- TxFreeQ.dWrite
        TMAC TDRFQ Read <- TxFreeQ.dRead
        TMAC TDRFQ End <- TxFreeQ.dEnd

    /* Tx Available Queue Initialization */
        dTxAvailablePhyQSize <- TxAvailableQ.dQSize + 1
        TxAvailableQ.dWrite <- TxAvailableQ.dStart
        TxAvailableQ.dRead <- TxAvailableQ.dStart + TxAvailableQ.dQSize
        TxAvailableQ.dEnd <- TxAvailableQ.dStart + dTxAvailablePhyQSize

    /* Put references on the Tx available queue */
        Use a FOR loop to assign references on the Tx available queue

        Initialize all rx descriptors to default values
        Initialize all tx descriptors to default values
        Initialize all the ChannelInfo pointers to NULL
        Initialize all the free block pointers to NULL
        Initialize the packet ID tables
        Initialize the interrupt context structures
        Configure serial links
        Configure GPIC interface
        Configure counters for performance monitor
        Configure RHDL and THDL
        Change the state of the FREEDM to INIT
        Return SUCCESS
    }
```

3.2.5 Activate FREEDM

After a successful master reset procedure, the FREEDM needs to be activated to start operation.

Function Prototype:

```
RC activateFREEDM(
   FREEDMContext *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be activated.
```

Return Value:

```
SUCCESS, or FAILURE if the FREEDM is not in INIT state.
```

Modified Value:

```
The state of the FREEDM, from INIT to ACTIVE.
```

Side Effects:

```
None.
```

Caveat:

```
The FREEDM should be initialized before the function activateFREEDM is
run.  The protocol software must have initialized the following fields
within the FREEDM context:

   BYTE  dCounterDelay;
   BYTE  dActivityDelay;
   BYTE  dProcessDelay;
```

Pseudo-Code:

```
RC activateFREEDM(
   FREEDMContext *pFreedm)
{
   If FREEDM is not in INIT state, set errno and return FAILURE
   Install the ISR for interrupt processing, if installHandles fails,
      return FAILURE
   Assign dIntEnable within the FREEDM context to FREEDM Master
      Interrupt Enable register
   RMAC Control register.ENABLE bit <- 1
   TMAC Control register.ENABLE bit <- 1
   If dCounterDelay is greater than 0, start timer which callback the
      pollCounter routine.  If startTimer fails, return FAILURE
   If dActivityDelay is greater than 0, start time which callback the
      pollActivity routine.  If startTimer fails, return FAILURE
   If dProcessDelay is greater than 0, start timer which callback the
      queueProcess routine.  If startTimer fails, return FAILURE
   Change the state of the FREEDM to ACTIVE
   Return SUCCESS
}
```

3.2.6 De-activate FREEDM

The state of FREEDM changes from ACTIVE to INIT after deactivation.

Function Prototype:

```
RC deactivateFREEDM(
   FREEDMContext *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be deactivated.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is not in ACTIVE state.
```

Modified Value:

```
The state of the FREEDM, from ACTIVE to INIT.
```

Side Effects:

```
None.
```

Caveat:

```
The FREEDM must be activated before this function is run.
```

Pseudo-Code:

```
RC deactivateFREEDM(
   FREEDMContext *pFREEDM)
{
   If FREEDM is not in ACTIVE state, set errno and return FAILURE
   FREEDM Master Interrupt Enable register <- 0x0
   Remove the ISR.  If removeHandlers fails, return FAILURE
   RMAC Control register.ENABLE bit <- 0
   TMAC Control register.ENABLE bit <- 0
   If dCounterDelay is greater than 0, stop the timer for polling.  If
      stopTimer fails, return FAILURE
   If dActivityDelay is greater than 0, stop the timer for polling.  If
      stopTimer fails, return FAILURE
   If dProcessDelay is greater than 0, stop the timer for queue
      processing.  If stopTimer fails, return FAILURE
   Change the state of the FREEDM to INIT
   Return SUCCESS
}
```

3.2.7 Provisioning

The receive and transmit channels are provisioned separately. Only one channel can be provisioned at a time.

Receive channels are provisioned as follow:

1. Set the CHDIS bit to 1 and write the channel number CHAN[6:0] that is being disabled to **RCAS Channel Disable 0x10C**.

2. Ensure that the BUSY bit of **RHDL Indirect Block Select 0x210** is set to 0 before programming channel FIFO. Set the block pointer value BPTR[8:0] in **RHDL Indirect Block Data 0x214**.

3. Writing the block number BLOCK[8:0] which owns the pointer to **RHDL Indirect Block Select 0x210**.

4. Repeat step 2 and 3 to form a circular linked list (with min. 3 blocks and max. 512 blocks).

5. Ensure that the BUSY bit of **RHDL Indirect channel Select 0x200** is set to 0 before a new indirect RAM access can be started. Set the PROV bit to 1 and assign one of the block number of the circular linked list as FPTR[8:0] to **RHDL Indirect Channel Data Register #1 0x204**. Setup CRC, STRIP, DELIN bit to any desired value.

6. Configure **RHDL Indirect Channel Data Register #2 0x208**.

7. Set the CRWB bit to 0 and write the channel number CHAN[6:0] to **RHDL Indirect channel Select 0x200**. Poll the BUSY bit to ensure it is 0 before proceeding.

8. Ensure that the BUSY bit of **RCAS Indirect Link and Time-slot Select 0x100** is set to 0 before provisioning RCAS block. Set the PROV bit to 1 and write the channel number CHAN[6:0] to **RCAS Indirect Channel Data 0x104**.

9. Set the RWB bit to 0 and write the link LINK[4:0] and the timeslot TSLOT[4:0] to **RCAS Indirect Link and Time-slot Select 0x100**.

10. In channelised links, if the channel spans multiple timeslots, register **RCAS Indirect Link and Time-slot Select 0x100** needs to be written once for each time-slot to associate to the channel. Ensure that the BUSY bit is 0 after each write.

11. Set the CHDIS bit to 0 and write the channel number CHAN[6:0] that is being enabled to **RCAS Channel Disable 0x10C** to indicate that provision is complete for that channel.

Function Prototype:

```
RC rxProv(
   FREEDMContext  *pFreedm,
   RxChannelInfo  *pRxProvInfo
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.

RxChannelInfo *pRxProvInfo:
   A pointer to the provision channel info structure to be used in
provisioning.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is not in ACTIVE state.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Caveat:

```
This function should be run in ACTIVE state.  All the fields in the
receive channel info must have initialized by the protocol software
before calling rxProv.
```

Pseudo-Code:

```
RC rxProv(
   FREEDMContext  *pFreedm,
   RxChannelInfo  *pRxProvInfo)
{
   If FREEDM is not in ACTIVE state, set errno and return FAILURE
   If the channel is provisioned, set errno and return FAILURE
   RCAS Channel Disable register <- channel to be disabled for prov
   Make the partial packet buffer list from free blocks and update the
      first block pointer within the FREEDM context
   If the BUSY bit in RHDL Indirect Channel Select register is not set
      to 0, poll the bit for a max period of 0.1 sec until it is set to
      0, otherwise set errno and return FAILURE
   Configure RHDL Indirect Channel Data #1 register
   Configure RHDL Indirect Channel Data #2 register
   RHDL Indirect Channel Select register <- channel to be provisioned
   If the BUSY bit in RHDL Indirect Channel Select register is not set
      to 0, poll the bit for a max period of 0.1 sec until it is set to
      0, otherwise set errno and return FAILURE
   If the BUSY bit in RCAS Indirect Link and Time-slot Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   RCAS Indirect Channel Data register <- channel to be provisioned
   RCAS Indirect Link and Time-slot Select register <- link and
      time-slot to be assigned to the provisioned channel, if the
      channel spans multiple time-slot, write register once for each
      time-slot and check BUSY bit after each write.
   RCAS Channel Disable register <- channel to be enabled after prov
   Update the channel info pointer within the FREEDM context
   Return SUCCESS
}
```

Transmit channels are provisioned as follow:

1. Set the CHDIS bit to 1 and write the channel number CHAN[6:0] that is being disabled to **TCAS Channel Disable 0x410**.

2. Set the block pointer value BPTR[8:0] in **THDL Indirect Block Data 0x3A4**.

3. Ensure that the BUSY bit is set to 0 in **THDL Indirect Block Select 0x3A0** before writing the block number BLOCK[8:0] which owns the pointer BPTW[8:0] (in **THDL Indirect Block Data 0x3A4**) to it.

4. To form a circular linked list, repeat step 2 and 3. This step is repeated as necessary (min. 3 blocks and max. 512 blocks).

5. Ensure that the BUSY bit is set 0 in **THDL Indirect Channel Select 0x380** before provisioning the THDL block. Set the PROV bit to 1 and set the FPTR[8:0] as one of the block number of the circular linked list in **THDL Indirect Channel Data #1 0x384**.

6. Configure **THDL Indirect Channel Data #2 0x388**.

7. Configure **THDL Indirect Channel Data #3 0x38C**.

8. Set the CRWB bit to 0 and write the channel number CHAN[6:0] to **THDL Indirect Channel Select 0x380** . Poll the BUSY bit to ensure it is 0 before proceeding.

9. Ensure that the BUSY bit in **TCAS Indirect Channel And Time-Slot Select 0x400** is set 0 before provisioning the TCAS block. Set the PROV bit to 1 and write the channel number CHAN[6:0] to **TCAS Indirect Channel Data 0x404**.

10. Set the RWB to 0 and write the link LINK[4:0] and the timeslot TSLOT[4:0] to **TCAS Indirect Channel And Time-Slot Select 0x400**.

11. In channelised links, if the channel spans multiple timeslots, registers **TCAS Indirect Channel And Time-Slot Select 0x400** needs to be written once for each time-slot to associate to the channel. Poll the BUSY bit until it is zero after each write.

12. Ensure that the BUSY is set 0 before setting the RWB to 0, setting the PROV bit to 1 and writing the channel number CHAN[6:0] to **TMAC**

Indirect Channel Provisioning 0x304. Poll the BUSY bit to ensure provisioning TMAC has completed.

13. Set the CHDIS bit to 0 and write the channel number CHAN[6:0] that is being enabled to **TCAS Channel Disable 0x410** to indicate that provision is completed for that channel.

Function Prototype:

```
RC txProv(
   FREEDMContext  *pFreedm,
   TxChannelInfo  *pTxProvInfo
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.

TxChannelInfo *pTxProvInfo:
   A pointer to the provision channel info structure to be used in
provisioning.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is not in ACTIVE state.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Caveat:

```
This function should be run in ACTIVE state.  All the fields in the
transmit channel info must have initialized by the protocol software
before calling txProv.
```

Pseudo-Code:

```
RC txProv(
   FREEDMContext  *pFreedm,
   TxChannelInfo  *ptxProvInfo)
{
   If FREEDM is not in ACTIVE state, set errno and return FAILURE
   If the channel has been provisioned, set errno and return FAILURE
   TCAS Channel Disable register <- channel to be disabled for prov
   Make the partial packet buffer list from free blocks and update the
      first block pointer within the FREEDM context
   If the BUSY bit in THDL Indirect Channel Select register is not set
      to 0, poll the bit for a max period of 0.1 sec until it is set to
      0, otherwise set errno and return FAILURE
   Configure THDL Indirect Channel Data #1 register
   Configure THDL Indirect Channel Data #2 register
   Configure THDL Indirect Channel Data #3 register
   THDL Indirect Channel Select register <- channel to be provisioned
   If the BUSY bit in THDL Indirect Channel Select register is not set
      to 0, poll the bit for a max period of 0.1 sec until it is set to
      0, otherwise set errno and return FAILURE
   If the BUSY bit in TCAS Indirect Link and Time-slot Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   TCAS Indirect Channel Data register <- channel to be provisioned
   TCAS Indirect Link and Time-slot Select register <- link and
      time-slot to be assigned to the provisioned channel
      If the channel spans multiple time-slots, write the register once
      for each time-slot and poll the busy bit to ensure it is 0 after
      each write
   If the BUSY bit in TMAC Indirect Channel Provisioning register is not
      set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE.
   TMAC Indirect Channel Provisioning register <- channel to be
      provisioned
   TCAS Channel Disable register <- channel to be enabled after prov
   Update the channel info pointer in the FREEDM context
   Return SUCCESS
}
```

3.2.8 Unprovisioning

The receive and transmit channels can be unprovisioned. The unprovisioned channels can be provisioned again provided that the block pointers are re-written to properly initialize the channel FIFO. Only one channel can be unprovisioned at a time.

Receive channels are unprovisioned as follow:

1. Set the CHDIS bit to 1 and write the channel number CHAN[6:0] that is being disabled to **RCAS Channel Disable 0x10C**.

2. Ensure that the BUSY bit in **RCAS Indirect Link and Time-slot Select 0x100** is set to 0 before unprovisioning the RCAS block. Set the PROV bit to 0 and write the channel number CHAN[6:0] to **RCAS Indirect Channel Data 0x104**.

3. Set RWB bit to 0 and write the link LINK[4:0] and the timeslot TSLOT[4:0] to **RCAS Indirect Link and Time-slot Select 0x100**.

4. In channelised links, if the channel spans multiple timeslots, registers **RCAS Indirect Link and Time-slot Select 0x100** needs to be written once for each time-slot to properly unprovision the channel. Poll the BUSY bit to ensure that it is 0 after each write.

5. Ensure that the BUSY bit of **RHDL Indirect Channel Select 0x200** is set to 0 before unprovisioning the RHDL block. Read the RHDL channel data by setting CRWB to 1 and writing the channel number CHAN[6:] to **RHDL Indirect Channel Select 0x200** .

6. Read **RHDL Indirect Channel Data Register #1 0x204** and proceed until the TAVAIL bit is zero.

7. Configure **RHDL Indirect Channel Data Register #1 0x204** and set the PROV bit to 0 in order to unprovision.

8. Set the CRWB to 0 and write the channel number CHAN[6:0] that needs to be unprovisioned to **RHDL Indirect Channel Select 0x200**. Poll the BUSY bit to ensure that it is 0 before proceeding.

9. Ensure that the BUSY bit is set to 0 before setting the RWB to 0, setting the PROV bit to 0 and writing the channel number CHAN[6:0] to **RMAC**

Indirect Channel Provisioning 0x284. Poll the BUSY bit to ensure it is 0 before proceeding.

10. Set the CHDIS bit to 0 and write the channel number CHAN[6:0] that is being enabled to **RCAS Channel Disable 0x10C** to indicate that unprovision is complete for that channel.

Function Prototype:

```
RC rxUnprov(
   FREEDMContext  *pFreedm,
   UBYTE          uRxUnprovChannel
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.

UBYTE uRxUnprovChannel:
   The receive channel to be unprovisioned.
```

Return Value:

```
RC (return code):
   SUCCESS, or FAILURE if the FREEDM is not in ACTIVE state.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Caveat:

```
This function should be run in ACTIVE state.
```

Pseudo-Code:

```
RC rxUnprov(
   FREEDMContext  *pFreedm,
   UBYTE          uRxUnprovChannel)
{
   If FREEDM is not in ACTIVE state, set errno and return FAILURE
   If the channel is unused, set errno and return FAILURE
   RCAS Channel Disable register <- channel to be disabled for unprov
   If the BUSY bit in RCAS Indirect Link and Time-slot Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   RCAS Indirect Channel Data register <- channel to be unprovisioned
   RCAS Indirect Link and Time-slot Select register <- link and
      time-slot to be assigned to the unprovisioned channel.  For
      channelised links, write the register once for each time-slots to
      be unprovisioned.  Poll the BUSY bit to ensure it is 0 after
      each write
   If the BUSY bit in RHDL Indirect Channel Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   Read the RHDL channel data by writing RHDL Indirect Channel Data #1
      register.  Poll the BUSY bit to ensure it is 0
   Read the RHDL indirect channel data and check that the TAVAIL bit is
      0 before proceeding
   RHDL Indirect Channel Data # 1 register <- set the PROV bit to 0
   RHDL Indirect Channel Select register <- channel to be unprovisioned.
      Poll the BUSY bit to ensure it is 0 before proceeding
   If the BUSY bit in RMAC Indirect Channel Provisioning register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   RMAC Indirect Channel Provisioning reg<- channel to be unprovisioned.
      Poll the BUSY bit to ensure it is 0 before proceeding
   RCAS Channel Disable register <- channel to be enabled after unprov
   Set NULL to the pointer pointing to this channel info structure
   Return SUCCESS
}
```

Transmit channels are unprovisioned as follow:

1. Ensure that the BUSY bit is set to 0 before setting the RWB bit to 0, setting the PROV bit to 0 and writing the channel number CHAN[6:0] to **TMAC Indirect Channel Provisioning 0x304**. Poll the BUSY bit to ensure that it is 0 before proceeding.

2. Set the CHDIS bit to 1 and write the channel number CHAN[6:0] that is being disabled to **TCAS Channel Disable 0x410**.

3. Ensure that the BUSY bit of **TCAS Indirect Channel And Time-Slot Select 0x400** is set to 0 before unprovisioning the TCAS block. Set the PROV bit to 0 and write the channel number CHAN[6:0] to be unprovisioned to **TCAS Indirect Channel Data 0x404**.

4. Set the RWB bit to 0 and write the link LINK[4:0] and the timeslot TSLOT[4:0] to **TCAS Indirect Channel And Time-Slot Select 0x400**. For channelised link, write the register once for each time-slot to be unprovisioned. Poll the BUSY bit to ensure it is 0 after each write.

5. Ensure that the BUSY bit of **THDL Indirect Channel Select 0x380** is set to 0 before unprovisioning the THDL block. Read the THDL channel data by setting CRWB to 1 and writing the channel number CHAN[6:] to **THDL Indirect Channel Select 0x380**. Poll the BUSY bit until it is 0 before proceeding.

6. Read **THDL Indirect Channel Data #1 0x384** and configure it by setting the PROV bit to 0.

7. Set the CRWB bit to 0 and write the channel number CHAN[6:0] to **THDL Indirect Channel Select 0x380**. Poll the BUSY bit to ensure that it is 0 before proceeding.

8. Set the CHDIS bit to 0 and write the channel number CHAN[6:0] that is being enabled to **TCAS Channel Disable 0x410** to indicate that unprovision is completed for that channel.

Function Prototype:

```
RC txUnprov(
   FREEDMContext  *pFreedm,
   UBYTE          uTxUnprovChannel
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.

UBYTE uTxUnprovChannel:
   The transmit channel to be unprovisioned.
```

Return Value:

```
RC (return code):
   SUCCESS, FAILURE if the FREEDM is not in ACTIVE state.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Caveat:

```
This function should be run in ACTIVE state.
```

Pseudo-Code:

```
RC txUnprov(
   FREEDMContext  *pFreedm,
   UBYTE          uTxUnprovChannel)
{
   If FREEDM is not in ACTIVE state, set errno and return FAILURE
   If the channel is unused, set errno and return FAILURE
   If the BUSY bit in TMAC Indirect Channel Provisioning register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   TMAC Indirect Channel Provisioning <- channel to be unprovisioned.
      Poll the BUSY bit until it is 0 before proceeding
   TCAS Channel Disable register <- channel to be disabled for unprov
   If the BUSY bit in TCAS Indirect Link and Time-slot Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   TCAS Indirect Channel Data register <- channel to be unprovisioned
   TCAS Indirect Link and Time-slot Select register <- link and
      time-slot to be assigned to the unprovisioned channel.  For
      channelised link, write the register once for each time-slot to be
      unprovisioned.  Poll the BUSY bit to ensure it is 0 after each
      write
   If the BUSY bit in THDL Indirect Channel Select register
      is not set to 0, poll the bit for a max period of 0.1 sec until it
      is set to 0, otherwise set errno and return FAILURE
   Read THDL Indirect Channel Data # 1 register by writing THDL Indirect
      Channel Select register
   Configure THDL Indirect Channel Data #1 register
   THDL Indirect Channel Select register <- channel to be unprovisioned.
      Poll the BUSY bit to ensure that it is 0 before proceeding
   TCAS Channel Disable register <- channel to be enabled after unprov
   Set NULL to the pointer pointing to this channel info structure
   Return SUCCESS
}
```

3.2.9 Transmit

This routine is responsible for writing a packet reference to the transmit ready queue. The status of the transmit request is not known until the TD reference is read from the transmit free queue. Therefore, the return code of this request routine is specified as PENDING. The confirmation of the transmit request is generated at a later time when the transmit confirm routine is called within the deferred processing routine.

Function Prototype:

```
RC transmit(
   FREEDMContext  *pFreedm,
   TxPacket       *pTxBuffer,
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.

TxPacket *pTxBuffer:
   A pointer to the link listed buffer to be transmitted.
```

Return Value:

```
RC (return code):
   PENDING, as the packet is not transmitted until the reference of the
transmit descriptor is read from the transmit free queue.  FAILURE if
the FREEDM is not in ACTIVE state or the software has run out of
available references.
```

Modified Value:

```
The write index of the transmit ready queue.
```

Side Effects:

```
None.
```

Caveat:

```
The FREEDM must be in the ACTIVE state before the function transmit
runs.  The protocol software must have initialized all the fields within
the tx packet before calling transmit.
```

Pseudo-Code:

```
RC transmit(
   FREEDMContext  *pFreedm,
   TxPacket       *pTxBuffer)
{
   If the FREEDM is not in ACTIVE state, set errno and return FAILURE
   If there is not enough references on the avail queue, set errno and
      return FAILURE
   Assign the info in the packet structure to the tx descriptor
   Queue the first descriptor reference of the packet on the transmit
      ready queue
   If the reference is not queued successfully, free the descriptors
      Back to the Tx available queue, set errno and return
      FAILURE
   Else return PENDING
}
```

3.3 Confirm and Indication Routines

The Confirm and Indication routines must be implemented within the protocol software. The FREEDM software calls these routines when it is in the ACTIVE state. Function prototypes for these routines are provided below:

3.3.1 Transmit Confirm

Routine confirmTx() is called after the software has read the transmit descriptor reference (TDR) from the Tx descriptor free queue and finished processing the TDR status.

```
void confirmTx( FREEDMContext *pFreedm, TxPacket *pTxBuffer);
/* *pFreedm- pointer to the FREEDM context                          */
/* *pTxBuffer- pointer to the Tx packet that has been transmitted */
```

3.3.2 Receive Indication

Routine indRx() is called after the software has read the receive packet descriptor reference (RPDR) from the Rx packet descriptor ready queue and finished processing the RPDR status.

```
void indRx( FREEDMContext *pFreedm, RxPacket *pRxBuffer);
/* *pFreedm- pointer to the FREEDM context                          */
/* *pRxBuffer- pointer to the Rx packet that has been received    */
```

3.3.3 Counter Status Indication

Routine indCountStat() is called when the PMON counters are polled.

```
void indCountStat( FREEDMContext *pFreedm);
/* *pFreedm- pointer to the FREEDM context                          */
```

3.3.4 Critical Error Indication

Routine indCriticalErr() is called when critical interrupts are generated.

```
void indCriticalErr( FREEDMContext *pFreedm, DWORD dCriticalInt);
/* *pFreedm- pointer to the FREEDM context                          */
/* dCriticalInt- value of the interrupt status register (for       */
/*    critical interrupts only)                                    */
```

4 PCI HARDWARE INTERFACE

4.1 Register Access Routines

The functions in the following sections are called by the FREEDM software to access the FREEDM registers:

4.1.1 Normal Register Access

Routine readRegister() is used to read normal mode registers of the FREEDM.

```
STATUS readRegister(int dDeviceHandle, int dRegister, DWORD *pValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- register offset                                       */
/* pValue- pointer to the value read from the register              */
```

Routine writeRegister() is used to write normal mode registers of the FREEDM.

```
STATUS writeRegister(int dDeviceHandle, int dRegister, DWORD dValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- register offset                                       */
/* dValue- value to be written to the register                      */
```

4.1.2 PCI Configuration Register Access

Routine readPCIRegisterDWord() is used to read individual dwords from the configuration space of the FREEDM.

```
STATUS readPCIRegisterDWord(int dDeviceHandle, int dRegister,
   DWORD *pValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* *pValue- pointer to the value read from the PCI configuration    */
/*    register                                                      */
```

Routine writePCIRegisterDWord() is used to write individual dwords from the configuration space of the FREEDM.

```
STATUS writePCIRegisterDWord(int dDeviceHandle, int dRegister,
   DWORD dRegisterValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* dRegisterValue- value to be written to the PCI configuration     */
/*    register                                                      */
```

Routine readPCIRegisterWord() is used to read individual words from the configuration space of the FREEDM.

```
STATUS readPCIRegisterWord(int dDeviceHandle, int dRegister,
   DWORD *pValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* *pValue- pointer to the value read from the PCI configuration    */
/*    register                                                      */
```

Routine writePCIRegisterWord() is used to write individual words from the configuration space of the FREEDM.

```
STATUS writePCIRegisterWord(int dDeviceHandle, int dRegister,
   DWORD dRegisterValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* dRegisterValue- value to be written to the PCI configuration     */
/*    register                                                      */
```

Routine readPCIRegisterByte() is used to read individual bytes from the configuration space of the FREEDM.

```
STATUS readPCIRegisterByte(int dDeviceHandle, int dRegister,
   DWORD *pValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* *pValue- pointer to the value read from the PCI configuration    */
/*    register                                                      */
```

Routine writePCIRegisterByte() is used to write individual bytes from the configuration space of the FREEDM.

```
STATUS writePCIRegisterByte(int dDeviceHandle, int dRegister,
   DWORD dRegisterValue);
/* dDeviceHandle- device handle number                              */
/* dRegister- PCI configuration register offset                     */
/* dRegisterValue- value to be written to the PCI configuration     */
/*    register                                                      */
```

4.2 Interrupt Service Routine

This function is implemented within the FREEDM software. It is called in response to an interrupt on the PCI bus. This function can only be called when the FREEDM software is in the ACTIVE state. An RTOS service call is made to install this ISR, and to remove this ISR.

This routine reads the interrupt status register and saves the interrupt status in the interrupt context structure for later interpretation by the Deferred Processing Routine (DPR).

Function Prototype:

```
BOOL FREEDMIsr(
   void *pIntContext
);
```

Function Arguments:

```
IntConext *pIntContext:
   A pointer to the interrupt context which contains information
necessary for interrupt processing.
```

Return Value:

```
TRUE if this PCI device has an active interrupt status, otherwise FALSE.
```

Modified Value:

```
pInterruptContext is updated with the interrupt status.
```

Side Effects:

```
Active interrupt status bits are disabled.
```

Caveat:

```
None.
```

Pseudo-Code:

```
BOOL FREEDMIsr(
   void *pIntContext)
{
   BOOL  bIsValidInt <- FALSE
   Read the FREEDM Master Interrupt Status register
   Mask off reserved bits and check if the interrupt is valid
   If valid, bIsValidInt <- TRUE
   Save the master interrupt status within the Interrupt context for DPR
   Disable interrupts
   Return bIsValidInt
}
```

5 RTOS SERVICES INTERFACE

5.1 Service Requests

The Freedm software calls the following RTOS service request functions:

5.1.1 Install/Remove Timer

```
DWORD installTimer();

STATUS startTimer( DWORD dTimerID, int dDelay, FuncPtr pFunc,
                   void *pArg);
/* dTimerID- ID of timer                                          */
/* dDelay- delay count, in ticks                                  */
/* pFunc- routine to call on time-out                             */
/* *pArg- pointer to parameter with which to call routine         */

STATUS stopTimer( DWORD dTimerID);
/* dTimeID- ID of timer                                           */

STATUS removeTimer( DWORD dTimerID);
/* dTimerID- ID of timer                                          */
```

5.1.2 Allocate/Deallocate Memory

```
void *malloc( DWORD dBytes);
/* dBytes- number of bytes to be allocated                        */

void free( void *memoryPtr);
/* *memoryPtr- pointer to the memory to be deallocated            */
```

5.1.3 Virtual/Physical Address Translation

```
DWORD mapVirtToPhys( DWORD uVirtualAddr);
/* uVirtualAddr- virtual address                                  */

DWORD mapPhysToVirt( DWORD uPhysAddr);
/* uPhysAddr- physcial address                                    */
```

5.1.4 Install/Remove ISR

```
STATUS installHandler(  DeviceHandle dFreedmHandle, FuncPtr pISRFunc,
                        FuncPtr pDPRFunc, void *pArg);
/* dFreedmHandle- device handle number                            */
/* pISRFunc- interrupt service routine to call upon interrupt     */
/* pDPRFunc- deferred process routine to call upon interrupt      */
/* *pArg- pointer to parameter with which to call ISR and DPR     */

STATUS removeHandler( DeviceHandle dFreedmHandle);
/* dFreedmHandle- device handle number                            */
```

5.1.5 Deferred Processing Routine

```
void FREEDMDPR( void *pIntContext);
/* *pIntContext- pointer to interrupt context                    */
```

5.1.6 Buffer Management

```
BYTE *getLgBuf();

BYTE *getSmBuf();

Buffer *getPtrBuffer();

TxPacket *getTxPacket();

RxPacket *getRxPacket();

STATUS putLgBuf( BYTE* pBufAddr);
/* *pBufAddr- pointer to the address of larget buffer          */

STATUS putSmBuf( BYTE* pBufAddr);
/* *pBufAffr- pointer to the address of small buffer           */

STATUS createIntContext(
         DeviceHandle dPCIHandle,DWORD dIntContextNumber);
/* dPCIHandle- device handle number                            */
/* dIntContextNumber- number of interrupt contexts to be allocated*/

BYTE *getIntContext ( DeviceHandle dPCIHandle);
/* dPCIHandle- device handle number                            */

STATUS putIntContext( DeviceHandle dPCIHandle, BYTE* pIntAddr);
/* dPCIHandle- device handle number                            */
/* *pIntAddr- pointer to the interrupt context to be put away  */

void deleteIntContext( DeviceHandle dPCIHandle);
/* dPCIHandle- device handle number                            */
```

5.2 Service Callbacks

These callback routines are installed during the service request. They are called upon timer expiration or interrupt.

5.2.1 Deferred Processing Routine

This function processes the interrupt status that was written to the interrupt context when servicing the PCI bus interrupt. This function is installed along with the interrupt service routine via the installHandler() RTOS service request.

Function Prototype:

```
VOID FREEDMDpr(
   void *pIntContext
);
```

Function Arguments:

```
IntContext *pIntContext:
   A pointer to the interrupt context which contains information
necessary for interrupt processing.
```

Return Value:

```
None.
```

Modified Value:

```
The interrupt context is being read after the DPR.
```

Side Effects:

```
None.
```

Pseudo-Code:

```
VOID  FREEDMDpr(
   void *pIntContext)
{
   /* interrupts are processed until there is no interrupt */
   A while loop to process interrupts until there is no interrupt
   Free the interrupt context after processing
   Enable the interrupts
}
```

```
VOID  intProcess{
   Void *pintContext)
{
   If the RPQRDYI interrupt bit is set
   Restart the process timer
   Process the rx ready queue and call the routine indRx for each
      packet reference read until the rx ready queue is empty
   Write the references to rx available queue for reuse

   If the TDQFI or IOCI interrupt bit is set
   Restart the process timer
   Process the tx free queue and call the routine confirmTx for each
      packet reference read until the tx free queue is empty
   Write the references to tx available queue for reuse

   If the RPQSFI interrupt bit is set
   Replenish the rx small free queue from the rx available queue with
      new buffers

   If the RPQLFI interrupt bit is set
   Replenish the rx large free queue from the rx available queue with
   new buffers

   If one of the severe interrupts have been reported i.e. SERRI, PERRI,
      RFCSEI, RABRTI, RPFEI, ROVRI, RPDFQEI, RPDRQEI, TDFQEI, TFUDRI,
      call the routine indCriticalErr
}
```

5.2.2 Counters Polling

This routine performs a constant polling of counters and modifies the values of counters within the FREEDM context strucutre. Counters are polled as follows:

- Read the count registers, **PMON Receive FIFO Overflow Count 0x504**, **PMON Receive FIFO Underflow Count 0x508**, **PMON Configurable Count #1 0x50C** and **PMON Configurable Count #2 0x510**.
- Read the **PMON Status 0x500** which indicates whether any of the four internal holding counters has overflowed.
- Write to **FREEDM Master Clock/BERT Activity Monitor and Accumulation Trigger 0x00C** to delimit the accumulation intervals in the PMON accumulation registers.

Function Prototype:

```
void  pollCounters(
   FREEDMContext  *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.
```

Return Value:

```
None.
```

Modified Value:

```
The counters values within the FREEDM context.
```

Side Effects:

```
None.
```

Pseudo-Code:

```
void  pollCounters(
   FREEDMContext  *pFreedm)
{
   Read the corresponding counters and save the values within the FREEDM
      context
   Read the PMON Status register and check if there is internal overflow
   Write to the FREEDM BERT Activity Monitor and Accumulation Trigger
      register to reload counters for next read
   Call the indCountStat routine
   Call the startTimer routine which call pollCounters routine at the
      expiration of the timer
}
```

5.2.3 Activity Polling

This routine is called periodically to perform a read from the FREEDM master clock activity monitor and the FREEDM master link activity monitor to detect for stuck at conditions. Values read from the registers are stored within the FREEDM context structure and processed by the protocol software.

Function Prototype:

```
void pollActivity(
   FREEDMContext  *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.
```

Return Value:

```
None.
```

Modified Value:

```
The activity monitors values within the FREEDM context.
```

Side Effects:

```
None.
```

Pseudo-Code:

```
void  pollActivity(
   FREEDMContext  *pFreedm)
{
   Read the FREEDM Master Clock Activity Monitor register and the
      FREEDM Master Link Activity Monitor register and store the values
      within the FREEDM context
   Call the startTimer routine which call pollActivity routine at the
      expiration of the timer.
}
```

5.2.4 Queues Processing

This routine processes the unread references on the receive ready queue and the transmit free queue at the process timer expiration. The timer restarts when RPQRDYI, TDQFI, or IOC interrupt occurs.

Function Prototype:

```
void  queueProcess(
   FREEDMContext  *pFreedm
);
```

Function Arguments:

```
FREEDMContext *pFreedm:
   A pointer to the context of FREEDM to be interrogated.
```

Return Value:

```
None.
```

Modified Value:

```
None.
```

Side Effects:

```
None.
```

Pseudo-Code:

```
void  queueProcess(
   FREEDMContext  *pFreedm)
{
   If the receive ready queue is not empty
   Process the rx ready queue and call the indRx routine for each packet
      reference read
   Write the references to rx available queue for reuse

   If the transmit free queue is not empty
   Process the tx free queue and call the routine confirmTx for each
      packet reference read
   Write the references to tx available queue for reuse

   Replenish the small/large rx free queue with new buffers

   Call the startTimer routine which call queueProcess routine at the
      expiration of the timer.
}
```

APPENDIX A: DATA TYPES AND CONSTANTS

The following data types and constants are defined in the header file:

```
typedef  char     BYTE;
typedef  uchar    UBYTE;
typedef  short    WORD;
typedef  ushort   UWORD;
typedef  long     DWORD;
typedef  ulong    UDWORD;

#define  MAX_CHANNELS   128
#define  MAX_BLOCKS     512
#define  MAX_LINKS      32
#define  MAX_DT_SIZE    16384
```

All variables that are used in this document follow a naming convention as shown in the table A1.

Table A1. Variable Type Prefixes

Variable Type	Prefix
Boolean	b
Pointer	p
Signed Integer	d
Unsigned Integer	u

NOTES

CONTACTING PMC-SIERRA, INC.

PMC-Sierra, Inc.
105-8555 Baxter Place Burnaby, BC
Canada V5A 4V7

Tel: (604) 415-6000

Fax: (604) 415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Application Information: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

PM-970280 (R2) Issue date: