Design description (DD)

9.0 Design approach

9.1 Overview

From the software communication interface's perspective, the CreditNet switch consists of the following major inter-processor interface attributes as illustrated in Figure 2 "System Block Diagram" on page 12.

1. A 16 Kbyte MCP <-> PCC Dual-Port memory (DP) on the Portcard - It provides a physical communication link between the PCC on the Portcard and the MCP on the MasterCard.
2. The MCP communicates with up to fifteen PCC each with their own DP memory as a physical link with the MCP. The DP memory are not shared between PCCs.
3. A 16 Kbyte PCC <-> SCH DP memory on the Portcard- It provides a physical communication link between the PCC and the Scheduler (SCH) processor which also resides on the Portcard.
4. Interrupts which may be used to inform a processor of service requests or arrival of messages. The interrupts available on the processors are:
   - one level interrupt from MCP to up to fifteen PCC.
   - two levels interrupt from PCC to MCP.
   - one level interrupt from PCC to SCH.
   - one level interrupt from SCH to PCC.
5. A EtherNet Interface between the Switch Workstation (WS) and the MCP.

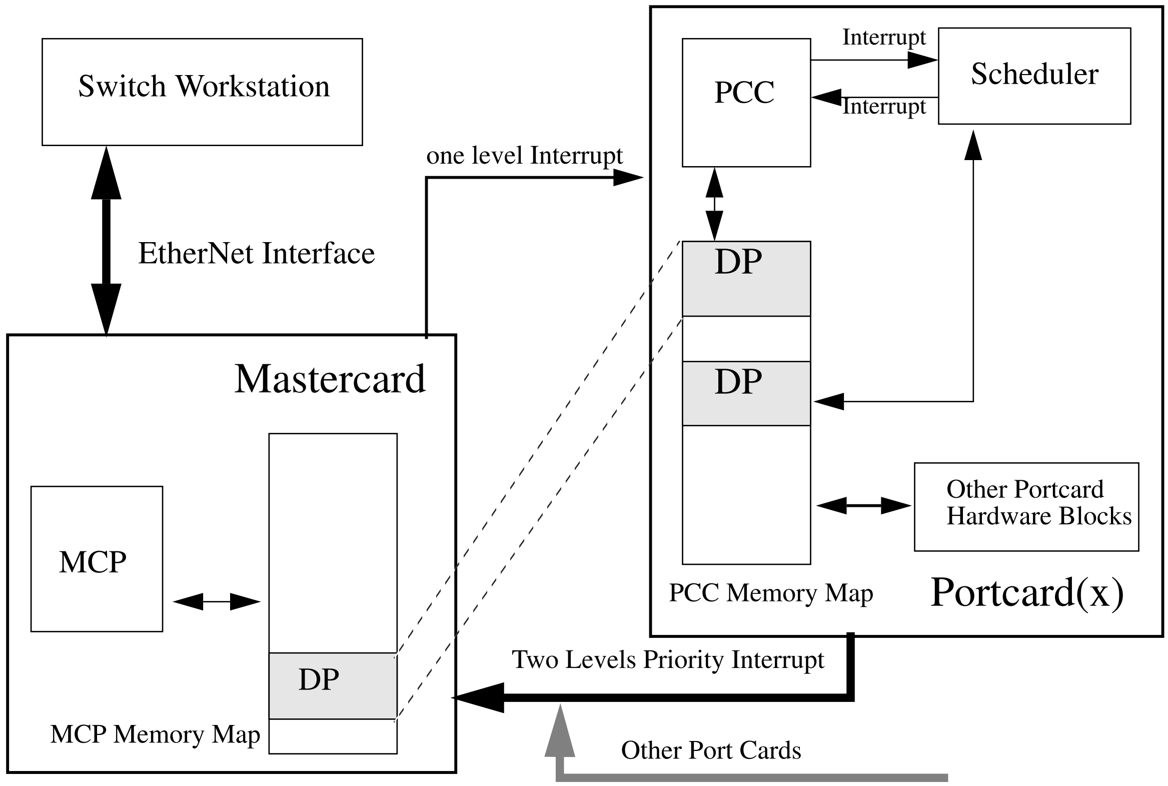

Figure 2 System Block Diagram

The following sections provide a detailed design description of software inter-processor communication interfaces for the CreditNet switch. These communication interfaces permit the following major functions:-

- Microprocessor software download from WS to PCC and SCH through MCP
- Switch and hardware configuration (including FLEX configuration data) download from WS to PCC and SCH through MCP.
- Messaging between MCP and PCCs
- Messaging between MCP and WS

The communication interfaces will be provided in the form of C functions and data structures in the following utility libraries:-

- MCP communication utility
- PCC communication utility

The design of the Communication Interfaces Software and the required messaging protocol are detailed in the following sections.

9.2 Detailed Description

9.2.1 PCC <-> MCP Communication Interface

The “messaging channels” between MCP and PCC essentially exist in the shared DP(s) between MCP and PCC on the PC as described earlier. Depending on how the shared memory is subdivided among various types and priorities of messages, multiple messaging channels can effectively be achieved. The communication protocol used for carrying messages and control information over the shared DP between MCP and PCC has the following attributes:

- A half-duplex protocol.
- The byte ordering scheme used is Little Endian.
- The use of checksum error detection.
- The option to disable error detection.
- The option to define data type to avoid potential word alignment problems since not all memory mapped locations are byte addressable.
- The use of a simple acknowledgment scheme.
- It enables bulk data downloading from MCP to PCC.
- It can support short operational messages.

9.2.1.1 Message Protocol

The general MCP <-> PCC message envelope is shown in Figure 3 "MCP <-> PCC message envelope" on page 13.

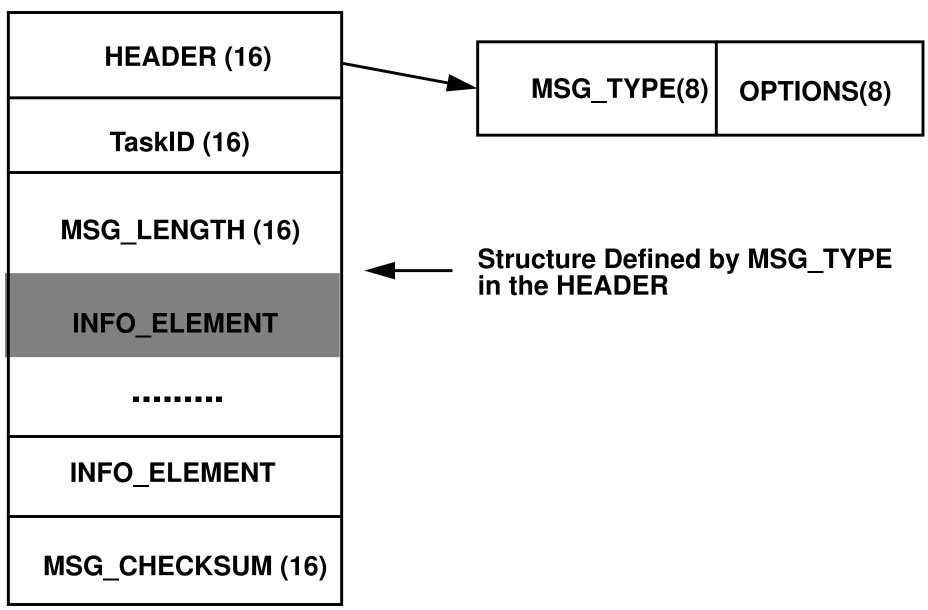

Figure 3 MCP <-> PCC message envelope

The message types and their corresponding opcodes are given in Table 2 "Opcodes for the MSG_TYPE sub-field" on page 14.

.

Table 2 Opcodes for the MSG_TYPE sub-field

OpCode	Message Type
0x00	WRITE
0x01	AUTONOMOUS
0x02	READ_REQUEST
0x03	READ_RESPONSE
0x04	BIT_SET_CLEAR
0x05	MONITOR_ADD
0x06	MONITOR_DELETE

- The data structure of the OPTION sub-field is shown in Table 3 "Bit definitions for the OPTION sub-field" on page 14. The MSG_CHECKSUM subfield is used to enable/disable message checksum. The DATA_TYPE subfield is used to define the data type of the contents in memory to be accessed. This is especially important in the case of non byte addressable, a fixed data type memory mapped hardware registers.

Table 3 Bit definitions for the OPTION sub-field

Bit Definition	OPTION
B0 ='0'	MSG_CHECKSUM error detection on data NOT required
B0 = '1'	MSG_CHECKSUM error detection on data required
B2,B1	DATA_TYPE (00) N/A (01) 8 bits (10) 16 bits (11) 32 bits

1 The taskID field is 16 bits wide. It is used to identify the receiver's functional block or server task at which the message is to be processed. the taskID will be set to zero in case of non Autonomous functions like (WRITE, READ_REQUEST).

2 The MSG_LENGTH field is 16 bits wide. It measures the length of the entire message in BYTES excluding the MSG_CHECKSUM field.

3 The INFO_ELEMENT fields contain the actual information of the message. The format is dependent on the message type.

4 The MSG_CHECKSUM field is16 bits wide. It contains the least significant 16 bits of the sum of the message contents starting from the HEADER field and ending at the last byte in the last INFO_ELEMENT. The summation is carried out byte wise. It is used to ensure integrity of the message received. MSG_CHECKSUM calculation and error detection may be enabled or disabled using the OPTION sub-field mentioned earlier. If it is disabled, however, MSG_CHECKSUM is defaulted to a value of 0xFFFF. For more information see "Checksum and data protection" on page 35..

9.2.1.2 Info Elements for different message types

9.2.1.2.1 WRITE

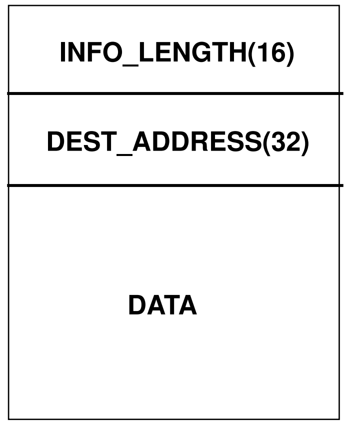

Figure 4 WRITE Info Element Format

The WRITE message results in some information being written into some memory/register addresses in the destination. The format of the message is as shown in Figure 4 "WRITE Info Element Format" on page 15.

1 The DEST_ADDRESS field is 32 bits wide and it defines the destination address to which data is written.

2 The INFO_LENGTH field is 16 bits wide and it defines the info element length including the 32 bits of the DEST_ADDRESS field.

3 DATA is the data to be written to the memory mapped to the receiving server.

9.2.1.2.2 AUTONOMOUS

Autonomous messages are unsolicited messages between the PCC and MCP. This is normally done in an asynchronous event such as discovery of a maintenance error or Virtual Channel (VC) setup. It is typically used for task invocation. The format of the message is shown in Figure 5 "Autonomous Info Element Format" on page 16.

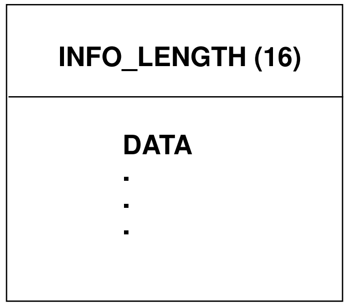

Figure 5 Autonomous Info Element Format

1 The INFO_LENGTH field is 16 bits wide and it defines the info element length.

2 The DATA is the autonomous message structure to be handled by the autonomous functions or tasks.

9.2.1.2.3 READ_REQUEST

The READ_REQUEST message results in some information being read from some memory/register addresses in the receiving end. The format of this message is shown Figure 6 "READ_REQUEST Info Element Format" on page 16.

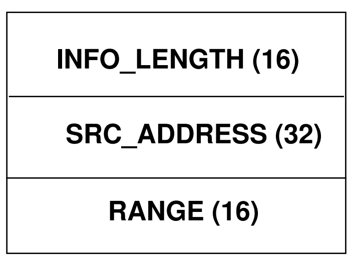

Figure 6 READ_REQUEST Info Element Format

This message normally results in the receiver end of the message responding with a READ_RESPONSE message type.

1 The SRC_ADDRESS field is 32 bits wide. It is used to identify the source address from which information are to be retrieved.

2 The INFO_LENGTH field is 16 bits wide and it defines the info element length including the 32 bits of the SRC_ADDRESS field and the 16 bit RANGE field.

3 The RANGE is length of data to be read from the memory mapped to the receiving server.

9.2.1.2.4 READ_RESPONSE

This message is sent in response to a READ_REQUEST message type. The format of this message is shown in Figure 7 "READ_RESPONSE Info Element Format" on page 17. The DATA contains the current memory contents requested by the READ_REQUEST message.

INFO_LENGTH (16)
DATA . .

Figure 7 READ_RESPONSE Info Element Format

9.2.1.2.5 BIT SET/CLEAR

The BITS SET/CLEAR messages cause one or more bits of a memory location/register to be changed in the destination. The message format is shown in Figure 8 "BIT SET/CLEAR Info Element Format" on page 17.

INFO_LENGTH (16)
DEST_ADDRESS (32)
RANGE (16)
AND MASK (32)
OR MASK (32)

Figure 8 BIT SET/CLEAR Info Element Format

1 The DEST_ADDRESS field is 32 bits wide. It specifies the destination address whose contents are to be modified.

2 The INFO_LENGTH field is 16 bits wide and it defines the info element length including the 32 bits of the DEST_ADDRESS, RANGE, AND_MASK and OR_MASK fields.

3 The RANGE field defines the number of consecutive locations of a given data type that need to be bit set or cleared starting from the DEST_ADDRESS.

4 The AND mask contains zeros at bit positions that must be cleared. The OR mask contains ones at bit positions that must be set.

9.2.1.2.6 MONITOR_ADD

This message is used to monitor the data memory/register location at the server end. The data to be monitored is periodically update data to a DP location. The message format is shown in Figure 9 "MONITOR_ADD_REQUEST Info Element Format" on page 18

INFO_LENGTH (16)
SRC_ADDRESS (32)
MONITOR_INDEX (16)

Figure 9 MONITOR_ADD_REQUEST Info Element Format

1 The SRC_ADDRESS field is 32 bits wide. It is used to identify the source address which is to be monitored periodically.

2 The MONITOR_INDEX is 16 bits wide. It defines the index to the Monitor List where the monitoring information are placed. For more information see "Monitor Data Structures" on page 34..

9.2.1.2.7 MONITOR_DELETE

INFO_LENGTH(16)
MONITOR_INDEX(16)

Figure 10 MONITOR_DELETE Info Element Format

This message is used by to inform the receiving server to stop monitoring a particular memory mapped location. The message format is shown in Figure 10 "MONITOR_DELETE Info Element Format" on page 18.

1 The MONITOR_INDEX is 16 bits wide. It defines the index to the Monitor List where monitoring information are placed For more information see "Monitor Data Structures" on page 34.

9.2.1.3 Operational Description

9.2.1.3.1 Downloading

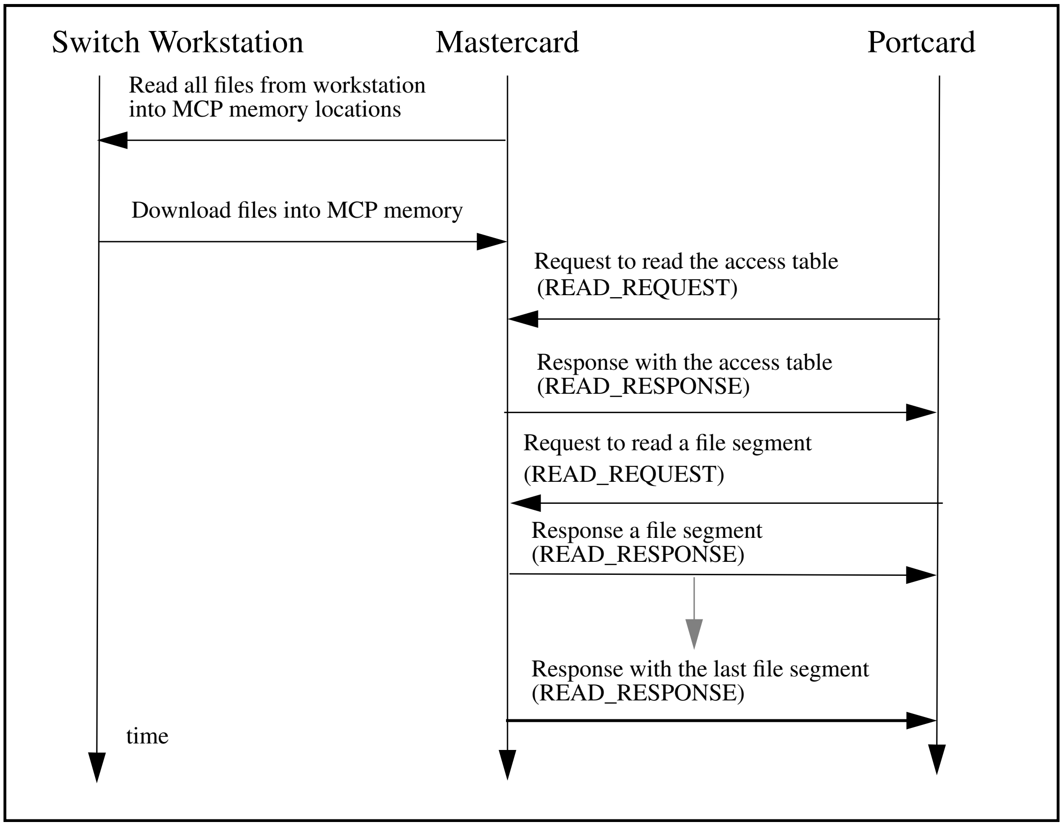

Figure 11 Scenario Diagram of downloading a file to the Portcard

During System Initialization the Mastercard needs to download files such as FLEX hardware configuration, software, initialization tables. etc. from the Switch Workstation to the PCC memory. in Figure 11 "Scenario Diagram of downloading a file to the Portcard" on page 19 shows how to download files from the Switch Workstation to the PCC memory:

1 MCP downloads all files from the Workstation host into its memory using VxWorks Remote File Access (NFS).

2 After loading files from the Workstation the MCP prepares an access table which is a list of memory pointers and sizes of those files in the MCP memory. The start address to that list is available to the PCC software.

3 PCC reads the access table from the MCP memory into its own static memory using READ_REQUEST and READ_RESPONSE messages.

4 PCC uses the access table to read those files into its own static memory using READ_REQUEST and READ_RESPONSE messages.

9.2.1.3.2 Logging Information

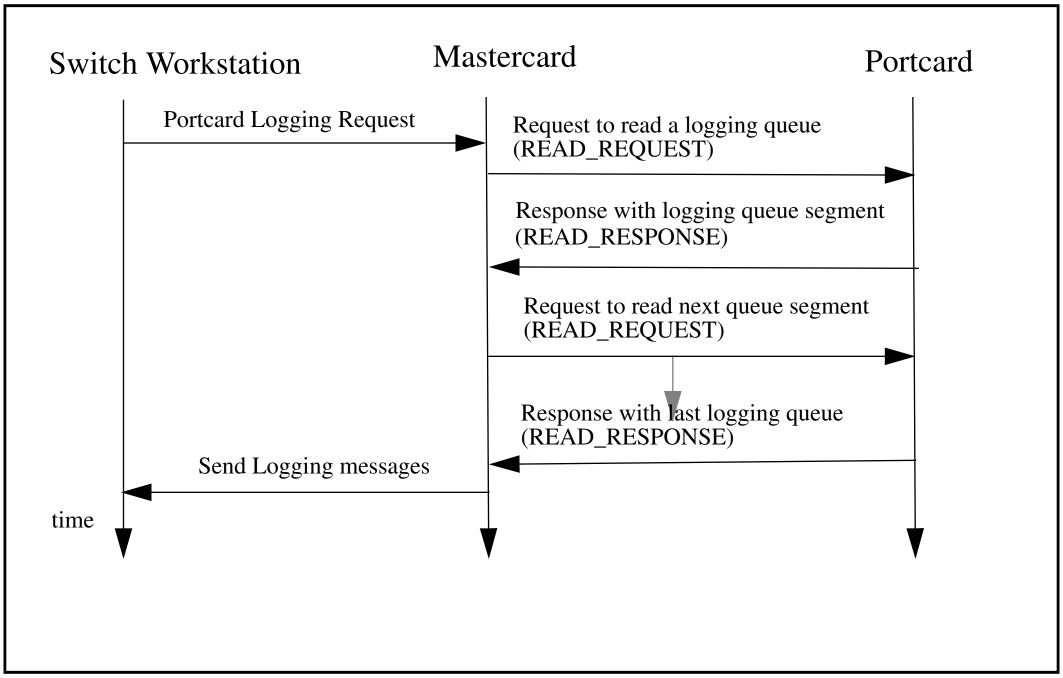

Figure 12 Scenario diagram of WS getting Portcard Logging informations

During Bootstrap, Initialization and Diagnostics tasks the UI needs to get logging information from the Portcards reporting their progress and status. This is illustrated in Figure 12 "Scenario diagram of WS getting Portcard Logging informations" on page 20:

1 PCC internally queues Portcard logging information.

2 MCP internally queues Mastercard logging information.

3 UI request Portcard(s) logging queue(s) from the MCP.

4 MCP reads Portcard(s) Logging queue using READ_REQUEST and READ_RESPONSE messages.

5 MCP is responsible for sending Mastercard and Portcard(s) logging information (in incoded form) to the WS and making it available to the UI for user display. VxWorks Logging tasks are used to send Logging information from the MCP to the WS.

6 UI uses the encoded logging information and a log message lookup table to display a readable log message.

9.2.1.3.3 Portcard Status Monitor

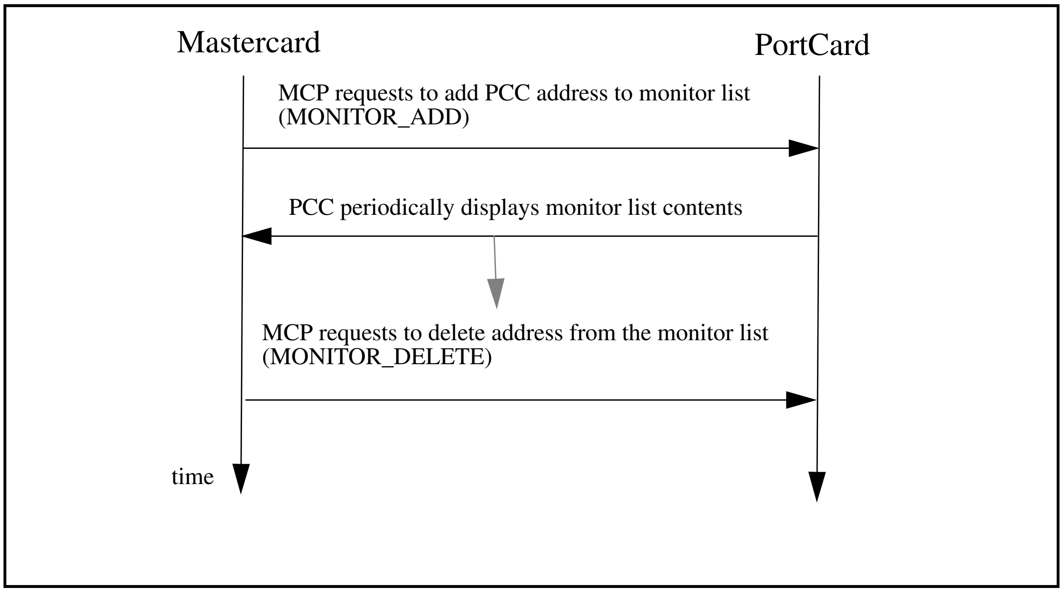

Figure 13 Scenario diagram of how the Mastercard monitors the Portcard

There are needs for the MCP to monitor some memory addresses mapped to the PCC and vice verse as illustrated in Figure 13 "Scenario diagram of how the Mastercard monitors the Portcard" on page 21.

1 MCP requests to add new PCC address to the monitor list using MONITOR_ADD message.

2 PCC periodically displays the contents of the monitor list in the DP shared memory.

3 MCP later can request to delete an entry from the monitor list using MONITOR_DELETE message.

9.2.1.3.4 Memory Access functions

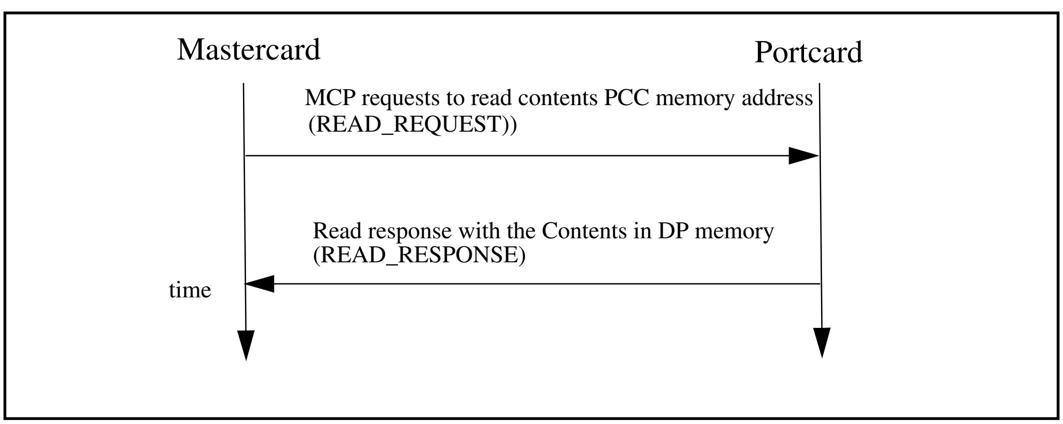

Figure 14 Scenario diagram of the MCP reading data from a PCC address

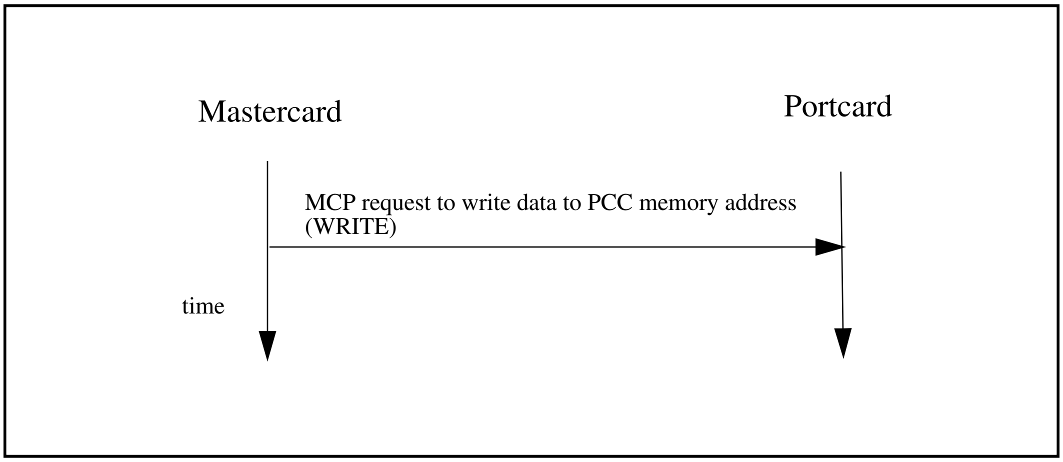

Figure 15 Scenario diagram of the MCP writing data to a PCC address

As illustrated in both Figure 14 "Scenario diagram of the MCP reading data from a PCC address" on page 22 and Figure 15 "Scenario diagram of the MCP writing data to a PCC address" on page 22 the Mastercard can issue a request to read from or write to an address mapped to the PCC.

The same operations are available to the PCC software to read and write an address mapped to the MCP.

There are two ways to invoke read and write operations, either by calling these functions inside the application, or interactively calling these functions from the VxWorks shell prompt from the workstation.

9.2.2 PCC to Scheduler Communication Interface

TBD

9.2.3 MCP <-> Switch Workstation Communication Interface

Communication Interface between the MCP and the Switch Workstation relies on the VxWorks Networking drivers building on top of (TCP/IP and UDP) protocols to provide the following major communication functions:

- Remote Access (NFS, FTP, TFTP)
- Remote Procedure Calls (RSH, NetDrv)
- Remote Login (rlogin, telnet)

For details refer to Chapter 6.2 (Network components) of [1].

9.2.4 Switch Workstation <-> PCC Communication Interface

The Communication interface between the Switch Workstation and the PCC uses the following methods:

1 The user can login on the MCP using a VxWorks shell session and use the communication interface functions interactively to establish, send, and receive messages from the PCC. This will be used mainly for debugging purposes.

2 The applications on the Switch Workstation may communicate to the PCC by sending messages to MCP task which works as a gateway between the PCC and the Workstation.

9.3 Design evolution/strategy

The inter-processor communication interface software is provided in the form of a utility library required to have the following design attributes:

- Based on client/server model to avoid potential real-time contention between processors.
- Supports both multi-thread and single-thread applications running on MCP and PCC processors (Design will accommodate both kinds of applications).
- Functionally Symmetric.
- Supports Remote Procedure Call (RPC).
- Reuse code as much as possible in both communication ends by coupling function.
- Supports priority levels.
- Supports addition of new functions.
- Data protection.

Here are the design procedure steps:-.

1 Define the basic functions and message protocol used by the Communication Interface and satisfy the previous features. Refer to Section "PCC <-> MCP Communication Interface" 9.2.1 on page 13 and Section "Operational Description" 9.2.1.3 on page 19 for more details.

2 Build communication methods on top of the basic functions to satisfy some application needs.

9.4 Modular structure

9.4.1 Communication Interface Architecture

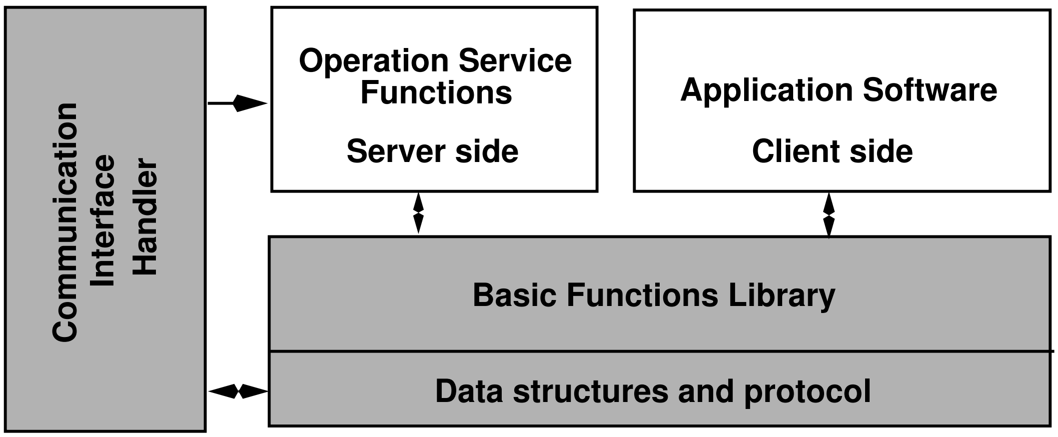

Figure 16 The Communication Interface Layers Diagram

The MCP <-> PCC communication interface is based on the client-server model as shown in Figure 16 "The Communication Interface Layers Diagram" on page 24

- The “non shaded” layers are the application layers where the client application software communicate with the Operation Service Functions in the server side using the communication interface layers.
- The “shaded” layers are the Communication Interface layers which control the communication channels between the client and the server and invoke the Server Operation Functions in the server side.

For more information see "Communication Interface Handlers" on page 30., "Operation Service Functions" on page 30 and "Basic Functions Library" on page 37.

9.4.1.1 Client/Server model

9.4.1.1.1 Channel Assignment

Channel assignment is based on the client-server model and a channel can be seized and freed by the client only. Each client server set requires a separate channel as illustrated in Figure 17 "The Communication channels between the MCP and the PCC" on page 25.

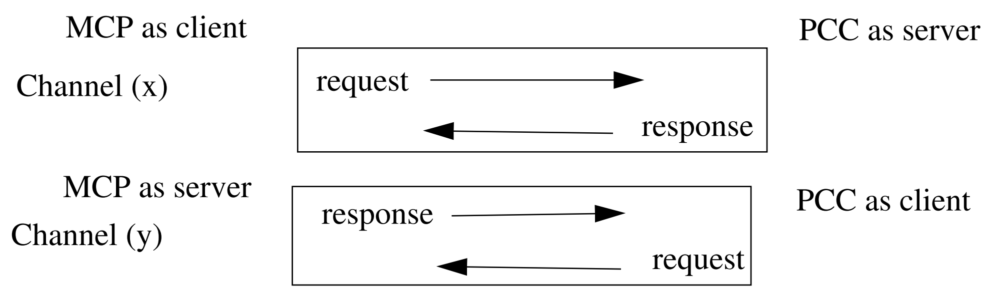

Figure 17 The Communication channels between the MCP and the PCC

This is to prevent the attempt by both MCP and PCC clients to allocate the same channel at the same time. For simplicity, there is no dynamic channel assignment. Instead, channels are pre-assigned for each (MCP/PCC) or (PCC/MCP) client/server set.

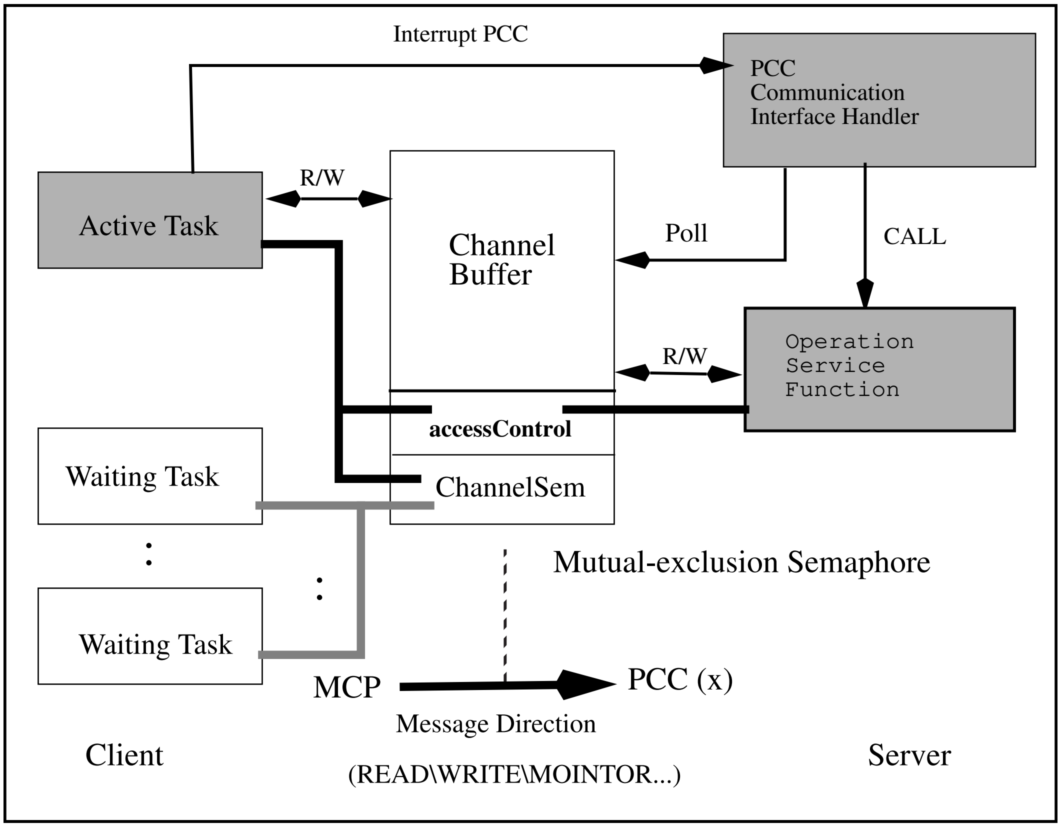

Figure 18 MCP-PCC Communication Interface

Figure 18 "MCP-PCC Communication Interface" on page 26 shows the MCP as a client and the PCC as a server.

- The MCP task which wants to seize a channel, takes the “ChannelSem” semaphore first and then uses the channel to communicate to the PCC.
- Any other tasks in the client side that want to use a seized channel will wait according to its priority until the channel is freed by the active task.
- The active MCP task can send messages to the PCC using the channel either by interrupting the PCC Communication Handler or waiting until the Handler periodically polls the channel(s) for waiting messages.
- When the Handler detects new message(s), it will dispatch them to the Operation Service Functions according to the message header.

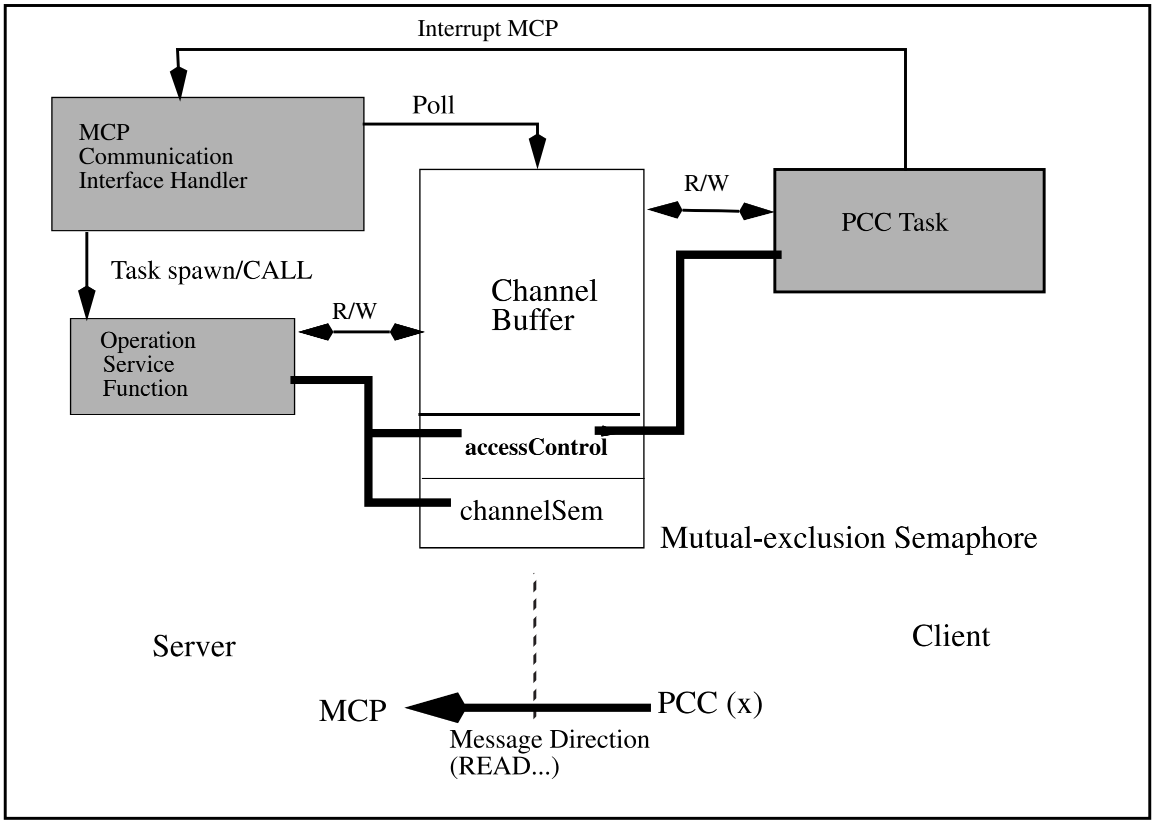

Figure 19 PCC-MCP Communication Interface

Figure 19 "PCC-MCP Communication Interface" on page 27 shows the PCC as a client and the MCP as a server. It is similar to the case of MCP as client and the PCC as server except that it takes advantage of VxWorks multi tasking kernel to spawn Autonomous Operation Service Functions as background tasks with different priorities.

9.4.1.1.2 Access Control

In order to regulate transfer of messages between MCP and PCC through the same channel in DP memory and to avoid contention in channel allocations between various tasks in either MCP or PCC, a 16-bit access control data structure is used as illustrated in Figure 20 "Access Control Data Structure Definition" on page 27.

ACCESS_CONTROL	ACCESS_CONTROL CHECKSUM
Upper Byte	**Lower Byte**

Figure 20 Access Control Data Structure Definition

The ACCESS_CONTROL data structure consists of two 8-bit fields: ACCESS_CONTROL, and ACCESS_CONTROL_CHECKSUM. The ACCESS CONTROL field is used to:-

- Indicate when a message can be sent.
- Indicate a message is being sent or a message is being received.

- Acknowledge message reception.
- Indicate checksum error during receiving.
- Indicate response is being sent.
- Indicate whether a message is from MCP to PCC or from PCC to MCP.

The ACCESS_CONTROL_CHECKSUM field is the one's complement of the ACCESS CONTROL field. It is used to capture potential corruption of the ACCESS CONTROL data. Table 4 "Access control state table" on page 29 shows the access control state table

Table 4 Access control state table

State	Next state(s)
Channel assigned to MCP client(s) is free Access Control = 0xF0	• MCP client set Access Control = 0xF1 to seize the channel.
Channel assigned to PCC client(s) is free Access Control = 0x0F	• PCC client set Access Control = 0x1F to seize the channel.
MCP client seized the channel Access Control = 0xF1	• MCP client set Access Control = 0xF0 to free the channel. • MCP client set Access Control = 0xF2 after sending a message to PCC server.
PCC client seized the channel Access Control = 0x1F	• PCC client set Access Control = 0x0F to free the channel. • PCC client set Access Control = 0x2F after sending a message to MCP server.
MCP client done sending a message Access Control = 0xF2	• PCC server set Access Control = 0xF4 to indicate checksum error during receiving a message. • PCC server set Access Control = 0xF8 after sending response message to MCP client. • PCC server set Access Control = 0xF1 after finish receiving a message with no response
PCC client done sending a message Access Control = 0x2F	• MCP server set Access Control = 0x4F to indicate checksum error during receiving a message. • MCP server set Access Control = 0x8F after sending response message to PCC client. • MCP server set Access Control = 0x1F after finish receiving a message with no response
PCC server found checksum error Access Control = 0xF4	• MCP client set Access Control = 0xF2 after message re-send. • MCP client set Access Control = 0xF1 to ignore checksum error.
MCP server found checksum error Access Control = 0x4F	• PCC client set Access Control = 0x2F after message re-send. • MCP client set Access Control = 0x1F to ignore checksum error.
PCC server response message to MCP Access Control = 0xF8	• MCP client set Access Control = 0xF2 to send next message • MCP client set Access Control = 0xF1 to indicate that no more messages.
MCP server response message to PCC Access Control = 0x8F	• PCC client set Access Control = 0x2F to send next message. • PCC client set Access Control = 0x1F to indicate that no more messages.

Figure 21 The Communication Interface state diagram

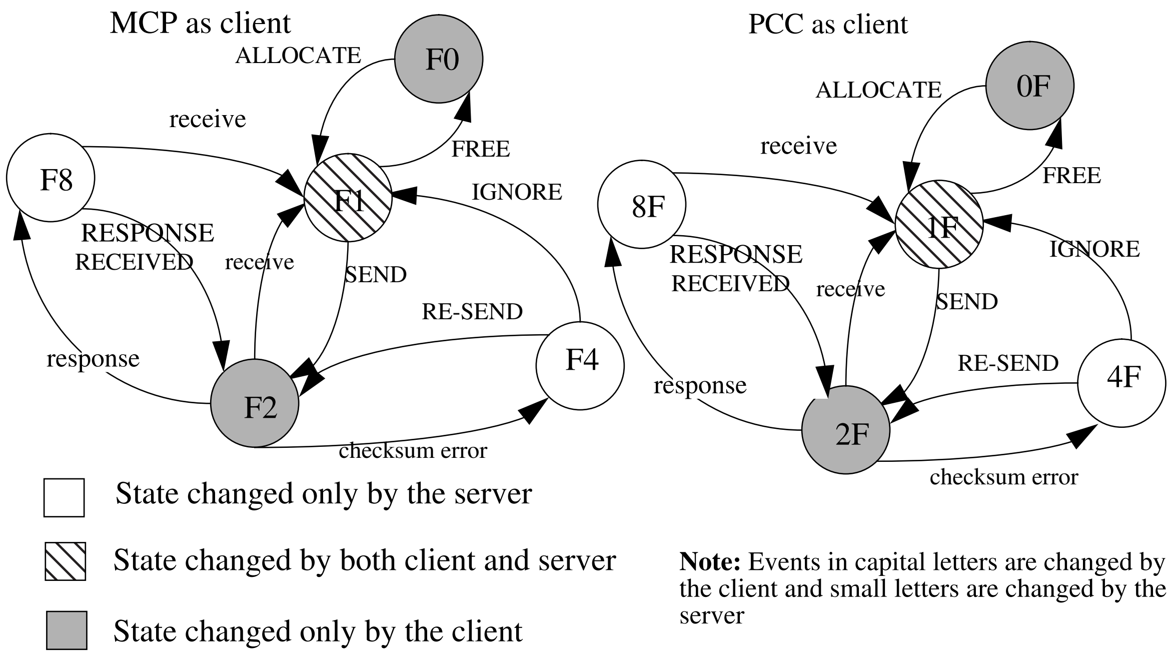

Figure 21 "The Communication Interface state diagram" on page 30 provides the state diagram of the communication Interface. The Access Control works as a state variable.

9.4.1.2 Communication Interface Handlers

This module mainly collects messages sent from the client(s) PCC or MCP to the Server(s) PCC or MCP and dispatches it according to the message header or TaskID if it is Autonomous message.

9.4.1.3 Operation Service Functions

These functions receive message(s) sent by the client and execute a certain function. There are two types:

General Functions: Handled by the Communication Interface Handler like (READ_REQUEST, READ_RESPONSE, MONITOR_ADD, etc.)

Autonomous Functions: Called or spawned by the Communication Interface Handler. But they are implemented by the application user in the server side to do a certain task like (VC setup etc.) those tasks internally may use some General Functions as previously defined.

9.4.2 Initialization

9.4.2.1 MCP to Switch Workstation Communication Establishment

The process of establishing the communication link between the MCP and the Switch Workstation over Ether network is part of the VxWorks Network Interface Drivers described in the BSP Porting Kit. For more information about porting these drivers refer to Part 9 (Creating a VxWorks Network Interface Driver) of [2].

9.4.2.2 (DP) Memory Initialization

There are some major data structures that need to be initialized on the DP before any of the Communication Interface functions start. Those structures are mainly used to store data which controls the Communication Interface operations. The PCC during bootstrap will allocate and initialize those data structures.

Following sections provide details about the following:

- Communication Interface Control Record.
- Channel(s) information.
- Channel(s) buffer.
- Monitor buffer.
- PCC monitor list.
- MCP monitor list

9.4.3 Data structures

Most of the Communication Interface data structures are in the DP shared memory which can be accessed by both MCP and PCC. Figure 22 "(DP) Memory map" on page 32 shows the DP memory map after being initialized by the PCC.

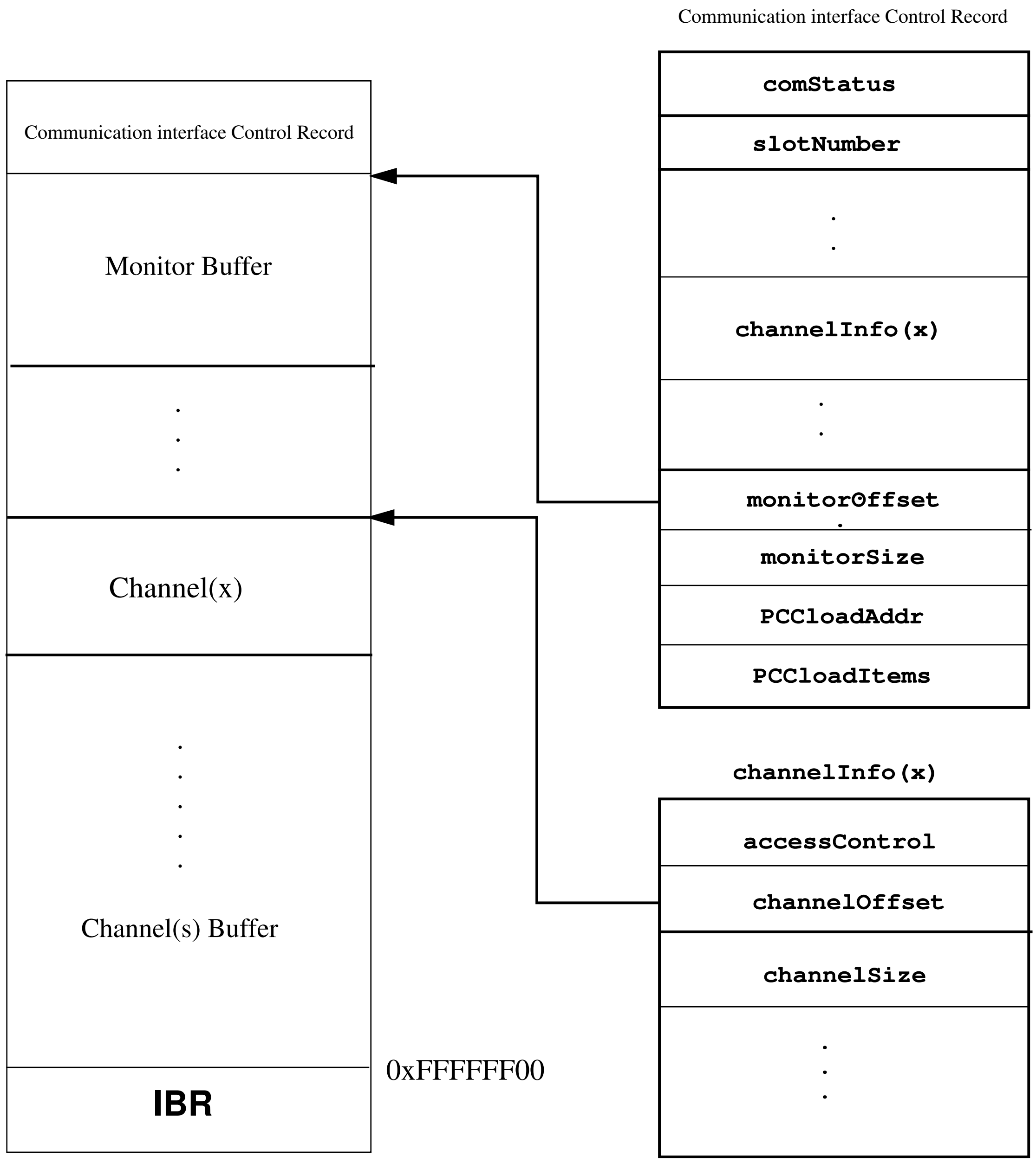

Figure 22 (DP) Memory map

9.4.3.1 IBR

The IBR is the primary data structure required to initialize the i960CA processor. For more details refer to Chapter 14 (Initialization And System Requirements) of [3].

9.4.3.2 The Communication Interface Control Record

The Control Record structure permits the following functions:

- Enables the Communication Interface to control its channel(s).
- PCC monitoring information.
- PCC downloading information.

```
#define MAX_CHANNEL 10
typedef struct
    {
    /* Channel(s) Information */
    int comStatus;                  /* Communication Interface status */
    char slotNumber;                /* Slot Number (TBD) */

    CHANNEL_INFO Channel[MAX_CHANNEL]; /* Array of channel(s) Information used
                                          to control the communication
                                          Interface */
    /* Monitor Information */
    char *monitorOffset;            /* Pointer to Monitor Buffer*/
    short mOnitorSize;              /* Monitor Maxsize */

    /* Download Information */
    char *PCCloadAddr;              /* PCC load image(s) start address */
    char PCCloadItems               /* PCC load image(s) items */
.
.
    }DP_CONTROL;
```

comStatus: This status flag used to enable or disable the communication interface functions. for re-initialization purposes and report communication interface error and states.

slotNumber: Identify the portcard slot number. The PCC use that number when it reports error to the MCP.

channelInfo[x]: This structure has all the per channel information:

```
typedef struct
      {
      Char channelSem;              /* Semaphore taken by client task */
      short accessControl;          /* Access Control*/
      short accessControl_B;        /* 1's comp of Access control */
      int OwnerID;                  /* Client task ID */
      int taskID;                   /* Copy of Autonomous Server task ID */
      int channelTimeout;           /* Time out value (0) if unset */
      short ChannelOffset;          /* Offset to channel Buffer where the
                                       messages are stored in the DP */
      short channelSize;            /* channel Buffer size */
      }CHANNEL_INFO;
```

- *ChannelSem:* Binary Mutual-Exclusion Priority Semaphore which to control the usage of a channel buffer and queues tasks which attempt to allocate the channel according to its priority.

- *accessControl*: Refer to Section "Channel Assignment" 9.5.1.1 on page 27
- ownerID: The client task ID which allocates the channel. This information is saved while allocating the channel and used in most of the functions to check that channels are only used by the owner client task.
- taskID: The autonomous server task ID which is invoked by the handler to perform a specific function on the server side. This field is equal to zero in case of basic functions like (READ_REQUEST, READ_RESPONSE, etc.).
- channelTimeout: when COM_TIME_OUT option is used in any of the comminution interface functions this value indicates how many ticks is allowed before the function returns time-out error.
- *channelOffset*: The offset of the channel buffer where messages are placed by the channel user.
- *channelsize:* The channel maximum size. The user can not send messages bigger than this size.

monitorOffset: The offset of the monitor area where the monitor items are periodically updated.

monitorSize: The monitor area size.

PCCloadAddr: The MCP start address where the PCC software and configuration files are stored after loaded from the Switch Workstation. The MCP set this value after initialization and make this information available to the PCC while bootstrap and download operation.

9.4.3.3 Monitor Data Structures

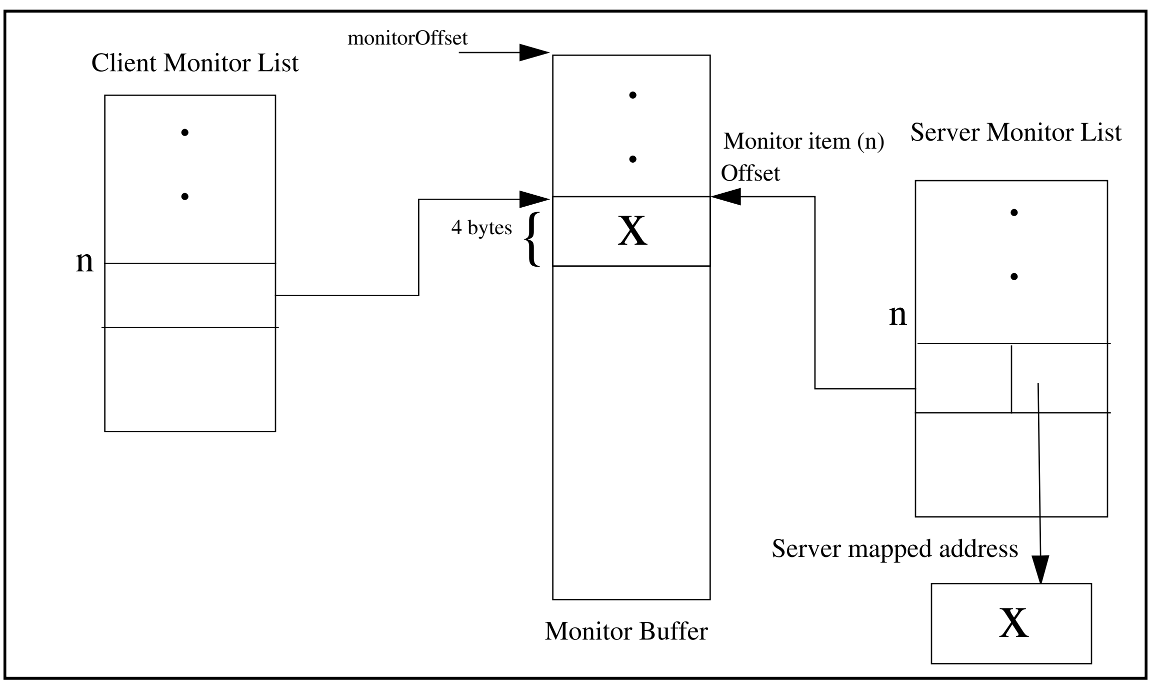

Figure 23 Monitor Data Structures

The monitor functions rely mainly on three major data structures:-

clientMonitorList: Each entry in this list has information which enables the client application software to have indirect access to a single monitor item. The list data structure resides in the client memory and is defined as follows.

```
#define MAX_MONITOR 50
typedef struct
    {
    char *monitorAddress;           /* DP monitor address where client can
                                       find the current value of monitor
                                       item.*/
    }MCP_MONITOR_STRUCT;

MCP_MONITOR_STRUCT clientmonitor_List[MAX_MONITOR];
```

Monitor Buffer: Memory buffer in the DP where the server periodically updates the current contents of the monitor items. Each item can take up to 4 bytes wide out of the Monitor buffer area.

serverMonitor List: This list is an image to the client monitor list but it also points to the server memory location subject to monitoring.

```
typedef struct
    {

    char *monitorAddress;           /* monitor address where PCC updates the
                                       value of the monitor item. */
    char *serverAddress;            /* PCC address subject to monitoring */

    }PCC_MONITOR_STRUCT;

PCC_MONITOR_STRUCT PCCmonitorList[MAX_MONITOR];
```

Figure 23 "Monitor Data Structures" on page 34 illustrates the relation between different data structures in the server and the client memory to control monitoring operations.

9.4.3.4 Channel Buffer

A memory pool where a fixed buffer is allocated for each channel. The amount of memory allocated for each channel is decided during system initialization. No de-allocation is allowed for that memory during the run time.

9.4.3.5 Checksum and data protection

This section covers different techniques used to protect data going between the client and server from being corrupted. The following are examples of some critical data structures that need to be protected:

- The access control variable corruption may lead to unexpected communication states.
- The DEST_ADDRESS corruption in the WRITE message leads to overwrites of undesired memory locations which may crash the whole portcard operations.

The main cause of data corruption is that the MCP writes to or reads from the DP over the backplane. The communication interface uses the following to protect data.

1 Data which is critical like channel “accessControl” is written to the DP bound with its 1’s complement “accessControl_B“. When another task wants to read this data the following check (accessControl XOR accessControl_B == 0) must be true.

2 The message envelope illustrated in Figure 3 "MCP <-> PCC message envelope" on page 13 is protected by 16 bits checksum which is the result of the byte wise addition of the whole message starting from the message header to the last byte in the last info element.Then the checksum is appended to the end of the message.

Note: The checksum calculation must be done on the message by the sender before it is actually written to the DP memory.

3 After the receiver read a message from the DP, it calculates the checksum as described above and compares the result with the stored checksum.

9.4.4 Application Interfaces

The Application uses the Communication Interface either by:

- Using a basic functions library which is the lowest level of interface the application can use.
- Using utility functions which are a higher level interface for some commonly used functions like MONITOR and READ/WRITE operations. This utility library is mainly for the client side.

Those functions are explained in the following sections in C language library definition. For more information see "Tutorial" on page 45. where there are some examples of how the client and server tasks can use the communication interface Library.

9.4.4.1 Linking communication Interface to application

TBD

9.4.4.2 Description of Basic Library

allocateChannel()

Name

allocateChannel()- Allocate and setup a Channel for communication.

External Interface

```
STATUS allocateChannel(port, channelID, ownerID, time-out, options)
    char port              /* Portcard Number */
    char *channelID        /* Channel Number */
    short ownerID          /* Client taskid */
    int time-out           /* time-out value to allocate a channel */
    char options           /* allocation options */
```

Purpose

- This function Allocates a channel to a client task.
- The client task must get its VxWorks task ID to make the channel remembers the owner task ID and prevent any other client tasks from using or freeing the channel using freeChannel().

- Any combination of the following options can be used when allocating a channel.

COM_WAIT_FOREVER	Wait forever till the channel become free.
COM_NO_WAIT	If channel not free do not wait.
COM_TIME_OUT	Time out value looking for a free channel
COM_GET_NEXT	If channel is not free, allocate the next available channel.

Returns

channelid: returns the first free channelid.
return value: OK or ERROR.

See Also

freeChannel()

freeChannel()

Name

freeChannel()- Close and free a Channel for communication.

External Interface

```
STATUS freeChannel(port, channelID, ownerID, options)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        TASK_ID ownerID         /*Client task ID which allocates the channel */
        char options            /* function options */
```

Purpose

- This function frees a channel and closes the connection between client and server.
- The client and server tasks can not send or receive messages any more using that channel.
- The next highest priority task that waits for the channel will automatically be allocated the channel.
- (options to be TBD)

Returns

return value: OK or ERROR

See Also

allocateChannel(), receiveMessage(), sendMessage();

getChannelTimeout()

Name

SetChannelTimeout()- Set time-out value for a channel.

External Interface

```
STATUS setChannelTimeout(port, channelID, ownerID, time-out)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        TASK_ID ownerID         /* Client task id which allocates the channel */
        int time-out            /* Time-out value for next operation on the channel */
```

Purpose

- This function sets the time-out value for the next operation on the channel.
- Operations like sendMessage(), receiveMessage(), allocateChannel() with COM_TIME_OUT options will return errors if the time-out value is expired before the function is completed.

Returns

return value: OK or ERROR

See Also

allocateChannel(), receiveMessage(), sendMessage()

tellChannelInfo()

Name

getChannelInfo() - Returns information about a channel

External Interface

```
STATUS getChannelInfo(port, channelID, info)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        CHANNEL_INFO *info)     /* Channel Information return structure */
```

Purpose

This function returns current information about a channel for debugging purposes.

Returns

info: Return structure contains all information about a channel
return value: OK or ERROR

sendMessage()

Name

sendMessage() - writes and sends a message to the server

External Interface

```
STATUS sendMessage(port, channelID, messagebuff, options)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        char *messagebuff       /* Message buffer */
        char options            /* Message write options */
```

Purpose

- The message is constructed first by the client task in the messagebuff and according to the message protocol.
- If messagebuff = NULL the client could construct the message directly in the channel Buffer before sending the message.
- If checksum option is chosen the function will calculate and add the checksum value to the message.
- After the message is written to the channel, the channel accessControl will change to notify the server that a new message is waiting.
- After the server task receives the message changes the accessControl and according to state table as illustrated in Table 4 "Access control state table" on page 29.
- The client has the option to wait (or time out waiting) for accessControl acknowledgment.
- The following options are available

COM_WAIT_FOREVER	Waits forever till message acknowledgment.
COM_NO_WAIT	Do not wait for acknowledgment.
COM_TIME_OUT	Waits till time-out expired.
COM_CHECK_SUM	Enables check sum for message contents.
COM_INTERRUPT	Message will interrupt the server.

Returns

return value: OK or ERROR

See Also

receiveMessage(), checkChannelStatus().

receiveMessage()

Name

receiveMessage()- Reads a message from a Channel.

External Interface

```
STATUS receiveMessage(port, channelID, messagebuff, options)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        char *messagebuff       /* Message buffer*/
        char options            /* Message writes options */
```

Purpose

- This function called by the server to read a message from a the channel,
- Perform checksum if required before the server gets the message
- Changes the message accessControl to acknowledge the client that the message is received ok or a checksum error is detected.

Returns

messagebuff: Return the message contents.
return value: OK or ERROR

See Also

sendMessage(), checkChannelStatus();

checkChannelStatus()

Name

checkChannelStatus()- checks current channel status

External Interface

```
STATUS checkChannelStatus(port, channelID, status);
    char port               /* Portcard Number */
    char channelID          /* Channel Number */
    char *status            /* Channel Status */
```

Purpose

This function checks the channel accessControl and returns its current value which is useful for polling message replies for the client side or message requests from the server side.

Returns

status: For more information see "Access control state table" on page 29.for different return values.
return value: OK, ERROR

See Also

sendMessage(), receiveMessage().

9.4.4.3 Description of Utility Library

writeRequest()

Name

writeRequest()- issue request to write an address mapped to the server

External Interface

```
STATUS writeRequest(port, channelID, address, buff, length, option);
    char port               /* Portcard Number */
    char channelID          /* Channel Number */
    char *address           /* Address needs to be written to */
    char *message           /* Buffer where the writ object is */
    char length             /* Data length*/
```

```
        char options              /* Write message options*/
```

Purpose

- This function is a higher level interface to the WRITE_REQUEST message.
- The function will construct the message envelope of WRITE_REQUEST message and add a new write request INFO_ELEMENT to the message.
- The last INFO_ELEMENT added using EOM option will send the message to the server.
- The following options are available.

COM_WAIT_FOREVER	Waits forever till message acknowledgment.
COM_NO_WAIT	Do not wait for acknowledgment.
COM_TIME_OUT	Waits till time-out expired.
COM_CHECK_SUM	Enables check sum for message contents.
COM_INTERRUPT	Message will interrupt the server
COM_EOM	End of write request messages.

Returns

return value: OK or ERROR

See Also

sendMessage()

readRequest()

Name

readRequest()- issues a request to write an address mapped to the server.

External Interface

```
STATUS readRequest(port, channelID, address, length, options);
        char port                 /* Portcard Number */
        char channelID            /* Channel Number */
        char *address             /* address needs to be read*/
        int length                /* Data length*/
        char options              /* Read message options */
```

Purpose

- This function is a higher level interface to the READ_REQUEST message.
- The function will construct the message envelope of READ_REQUEST message and then add a new INFO_ELEMENT to the message.
- The last INFO_ELEMENT added using EOM option will send the message to the server.
- The following options are available.

COM_WAIT_FOREVER	Waits forever till message acknowledgment and READ_RESPONSE comes back.
COM_NO_WAIT	Do not wait for acknowledgment or read response.
COM_TIME_OUT	Waits till time-out expired.
COM_CHECK_SUM	Enables check sum for message contents.
COM_INTERRUPT	Message will interrupt the server
COM_EOM	End of read requests messages.

Returns

return value: OK or ERROR

See Also

sendMessage()

readResponse()

Name

readResponse() - reads a read response info element

External Interface

```
STATUS readResponse (port, channelID, responseBuffer, element, status, options)
char port                   /* Portcard Number */
char channelID              /* Channel Number */
char *responseBuffer        /* Response address */
char *element               /* Info Element */
char *status                /* (1) initial value */
                            /* (2) next message */
                            /* (0) last message */
char options                /* response Options */
```

Purpose

- This function receives a read response info element previously requested by readRequest().
- The (responseBuffer) is a Buffer allocated by the client to contain all read response info elements. If (responseBuffer) is not equal to NULL the readResponse() will initially receive and copy all info elements from the channel buffer into the (responseBuffer) else the response will remain in the channel buffer.
- The (Status) must be initialized to the value (1). then the (status) value change to (2) when readResponse() returns info element(s). The last element will change the (status) to value (0).
- (element) is a pointer to the current read response info element.

Returns

return value: OK or ERROR
status: (0) last message, (1) initial value, (2) next element.

See Also

readRequest()

bitSetClear()

Name

bitSetClear()- issue request to set/clear bits an address mapped to the server

External Interface

```
STATUS bitSetClearIMPORT(port, channelID, address, ANDmask, ORmask, length,
options);
    char port              /* Portcard Number */
    char channelID         /* Channel Number */
    char *address          /* Address needs to be set */
    short ANDmask          /* AND mask */
    short ORmask           /* OR mask */
    char length            /* Data length */
    char options           /* Bit set clear message options */
```

Purpose

- This function is a higher level interface to the BIT_SET_CLEAR message.
- The function will construct the message envelope and add a new INFO_ELEMENT to the message.
- The last INFO_ELEMENT added using EOM option will send the message to the server.
- The following options are available

COM_WAIT_FOREVER	Waits forever till message acknowledgment,
COM_NO_WAIT	Do not wait for acknowledgment.
COM_TIME_OUT	Waits till time-out expired.
COM_CHECK_SUM	Enables check sum for message contents.
COM_INTERRUPT	Message will interrupt the server
COM_EOM	End of write request messages.

Returns

return value: OK or ERROR

See Also

sendMessage()

addMonitor()

Name

addMonitor()- adds a new monitor address mapped to the server

External Interface

```
STATUS addMonitor(port, channelID, monitorAddress, monitorIndex);
    char port               /* Portcard Number */
    char channelID          /* Channel Number */
    char *monitorAddress    /* Address needs to be monitored */
    int *monitorIndex       /* Returns the MCP monitor access table index */
```

Purpose

This function requests to add a new address mapped to the server to be monitored.

Returns

monitorIndex: returns the index to the MCP monitor access table which is used by the MCP software to access the current monitor value.

return value: OK or ERROR

See Also

sendMessage(), deleteMonitor()

deleteMonitor()

Name

deleteMonitor()- deletes a monitor item

External Interface

```
STATUS deleteMonitor(port, channelID, monitorIndex);
    char port               /* Portcard Number */
    char channelID          /* Channel Number */
    char monitorIndex       /* MCP monitor access table index */
```

Purpose

- This function deletes a monitor to an address mapped to the server.
- The monitorIndex is the index to client monitor access table for the address need to be deleted and stop the server from monitoring it.

Returns

return value: OK or ERROR

See Also

sendMessage(), addMonitor()

sendAutonomous()

Name

sendAutonomous() - sends autonomous message to the server

External Interface

```
STATUS sendAutonomous(port, channelID, taskID, messagebuff, length, options)
        char port               /* Portcard Number */
        char channelID          /* Channel Number */
        int taskID              /* Autonomous server task ID */
        char *messagebuff       /* Message buffer*/
        short length            /* Message Length */
        char options            /* Message write options */
```

Purpose

- Client task constructs the autonomous info element into (messagebuff) buffer.
- The (taskID) is the taskID of the server task which receives and executes the corresponding service function to the autonomous message.
- The (length) is the length of the autonomous message.
- The following options are available

COM_WAIT_FOREVER	Waits forever till message acknowledgment.
COM_NO_WAIT	Do not wait for acknowledgment.
COM_TIME_OUT	Waits till time-out expired.
COM_CHECK_SUM	Enables check sum for message contents.
COM_INTERRUPT	Message will interrupt the server
COM_EOM	End of Autonomous messages.

Returns

return value: OK or ERROR

See Also

sendMessage()

9.4.4.4 Tutorial

The following sections will explain in C language code how the application may interface to Communication Interface Functions.

9.4.4.4.1 Using General messages

The following code example depicts a task executing on MCP (as a client) and PCC (as server), The client task uses the Communication Interface to send read request messages to the server task running in the PCC.

The following code is executed by the client task on the MCP:

```
/* clientTask1.c - general client example */

#include "vxWorks.h";
#include "stdio.h";
#include "taskLib.h";
#include "comIF.h";

/*****************************************************************
*clientTask1 - MCP client task
*
*
*/

STATUS clientTask1()
    {
    int ownerID;
    char channelID=0;
    int dumdata[5];
    char status=1;

    /* Get current task ID */

    ownerID = taskIdself();

    /* Allocates any communication channel to Portcard (0) */

    if (allocateChannel(0, &channelID, ownerID, 20,COM_TIME_OUT) == ERROR)
      return(ERROR);

    /* Now request to read some PCC addresses to the channel which will
      interrupt the PCC*/

    if (readRequest(0, channelID, I_EPATH_BUS_XX, 10, 0) == ERROR)
      return(ERROR);

    if (readRequest(0, channelID, I_EPATH_BUS_YY, 10, 0) == ERROR)
      return(ERROR);

    if (readRequest(0, channelID, I_EPATH_BUS_ZZ, 10, COM_TIME_OUT|COM_EOM)
                    == ERROR)
      return(ERROR);
    .
    .
    .
    /* The read response message is now in the channel, go and get it*/
    while (status == 1)
      {
      if(readResponse(0,channelID, NULL, (char*)&dumdata[0], &status,
                           COM_TIME_OUT) == ERROR)
        return(ERROR);
      }
    .
    .
/* If you do not need the channel any more free it */
    freeChannel(0, channelID, ownerID, 0);

    }
```

9.4.4.4.2 Using Autonomous Messages

The following code example depicts two tasks executing on MCP (as a client) and PCC (as server), The client task uses the Communication Interface to send autonomous messages to the server task running in the PCC.

The following code is executed by the client task on the MCP:

```
/* clientTask2.c - general autonomous server example */

#include "vxWorks.h"
#include "stdio.h"
#include "taskLib.h";
#include "comIF.h";

/*****************************************************************
*clientTask1 - MCP client task
*
*
*/

STATUS clientTask2()
    {
    int ownerID;
    char channelID=0;
    USER1_STRUCT user1Message;
    char reply[100];

    /* Get current task ID */

    ownerID = taskIdself();

    /* Allocates any communication channel to Portcard (0) */

    if (allocateChannel(0, &channelID, ownerID, 20, COM_TIME_OUT) == ERROR)
      exit(ERROR);
    .
    .
    /* The user has already constructed the message in user1Message */
    If (sendMessage(0, channelID, &user1Message, COM_TIME_OUT) == ERROR)
      exit(ERROR);
    .
    /* Now check for a reply from the server */
    if (receiveMessage(0, channelID, &reply[0], COM_TIME_OUT)== ERROR)
      exit(ERROR);
    .
    /* Now check the reply for whatever it is */
    .
/* If you do not need the channel any more free it*/
    freeChannel(0, channelID, ownerID, 0);
  }
```

The following code is executed by the server task on the PCC:

```
/* serverTask2.c - general client example */

#include "vxWorks.h";
#include "stdio.h";
#include "taskLib.h";
```

```
#include "comIF.h";

/*****************************************************************
* serverTask2 - PCC server task
* This task will be spawned and deleted by the communication Interface
* Handler in the server side (PCC)
*/

STATUS serverTask2(char channelID)
    char channelID; /* channel number which has a message for that autonomous
                       server */
    {
    int taskID;
    USER1_STRUCT_REPLY user1Reply;
    char Message[100];


    while(TRUE){
      /* Waiting for a message from the client task */
      If (receiveMessage(0, channelID, &Message[0], COM_WAIT_FOREVER) ==
                         ERROR)
          return(ERROR);
    .
    .
      /* Do the required service function and construct a reply message */
    .
    .
      /* Now reply to the server */
      if (sendMessage(0, channelID, &user1Reply, COM_WAIT_FOREVER)== ERROR)
          return(ERROR);

    }

  }
```

9.4.4.4.3 Using monitor functions

The following code example depicts a task executing on MCP (as a client) and needs to monitor memory addresses mapped to the PCC (as server), The client task uses the Communication Interface to add monitor, check its value and then delete that monitor whenever its is no longer needed.

The following code is executed by the client task on the MCP:

```
/* clientTask3.c - general client example */

#include "vxWorks.h";
#include "stdio.h";
#include "taskLib.h";
#include "comIF.h";

#define MONITOR_ITEM(x) *(MCPmonitorist[x].monitorAddress)

/*****************************************************************
*clientTask3 - MCP client task
*
*
*/

STATUS clientTask3()
```

```
{
int ownerID;
char channelID=0;
int monIndex;

/* Get current task ID */

ownerID = taskIdself();

/* Allocates any communication channel to Portcard (0) */

if (allocateChannel(0, &channelID, ownerID, 20, COM_TIME_OUT) == ERROR)
  exit(ERROR);

.
.
/* Now request to add a monitor to PCC addresses */
if (addMonitor(0, channelID, I_EPATH_BUS_XX, &monIndex) == ERROR)
  return(ERROR);


/* How to Check the monitor value */
if(MONITOR_ITEM(monIndex) > 1000)
  return(ERROR);
.
.
/* Monitor is no longer needed. Delete the monitor*/
deleteMonitor(0,channelID,monIndex);
.
.
.
/* the channel no longer needed. Free it*/
freeChannel(0, channelID, ownerID, 0);

}
```

10.0 Real time estimates

11.0 Glossary of definitions and abbreviations

DP	Dual Port Memory.
FIFO	First In First Out.
FTP	File Transfer Protocol.
IBR	
IP	Internet Protocol.
LAN	Local Area Network.

MCP	Master Card Processor.
NFS	Network File System.
PCC	Port Card Controller.
RPC	Remote Procedure Call.
RSH	Remote Shell.
TCP/IP	Transmission Control Protocol / Internet Protocol.
TFTP	Trivial File Transfer Protocol.
VC	Virtual Circuit.
WS	Switch Workstation.

12.0 References

[1] "VxWorks Programmer's Guide" Release 5.1 - Wind River Systems

[2] "VxWorks BSP Porting Kit" Release 5.1 - Wind River Systems

[3] i960 CA Microprocessor User Manual - intel