PXI™-8

# PXI MultiComputing Software Specification

Revision 1.1
May 31, 2018

PXI Systems Alliance

# IMPORTANT INFORMATION

## Copyright



## NOTICE

The *PXI MultiComputing (PXImc) Software Specification* is authored and copyrighted by the PXI Systems Alliance. The intent of the PXI Systems Alliance is for the *PXImc Software Specification* to be an open industry standard supported by a wide variety of vendors and products. Vendors and users who are interested in developing PXI-compatible products or services, as well as parties who are interested in working with the PXI Systems Alliance to further promote PXI as an open industry standard, are invited to contact the PXI Systems Alliance for further information.

The PXI Systems Alliance wants to receive your comments on this specification. Visit the PXI Systems Alliance web site at `http://www.pxisa.org/` for contact information and to learn more about the PXI Systems Alliance.

The attention of adopters is directed to the possibility that compliance with or adoption of the PXI Systems Alliance specifications may require use of an invention covered by patent rights. The PXI Systems Alliance shall not be responsible for identifying patents for which a license may be required by any PXI Systems Alliance specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. PXI Systems Alliance specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

The information contained in this document is subject to change without notice. The material in this document details a PXI Systems Alliance specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

The PXI Systems Alliance makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The PXI Systems Alliance shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Compliance with this specification does not absolve manufacturers of PXI equipment from the requirements of safety and regulatory agencies (UL, CSA, FCC, IEC, etc.).

## Trademarks

PXI™, PXI-Express, PXImc, the PXImc Logo, and the PXImc Glyph are trademarks of the PXI Systems Alliance.

PICMG™ and CompactPCI® are trademarks of the PCI Industrial Computation Manufacturers Group.

Product and company names are trademarks or trade names of their respective companies.

# PXI MultiComputing Software Specification Revision History

This section is an overview of the revision history of the *PXI MultiComputing (PXImc) Software Specification*.

## Revision 1.0 RC 1, September 16, 2009

This is the first public revision of the *PXI MultiComputing Software Specification*.

## Revision 1.1, May 31, 2018

Added text for PXIMC_enableDeviceAccess.

Changed filenames for the PXImc Dispatcher on the Linux operating system.

This Page Intentionally Left Blank.

# Contents

# 1. Introduction

The focus of the *PXI MultiComputing (PXImc) Software Specification* is to define a shared memory and other standard protocols to allow CPU Based Devices to communicate through Non-Transparent Bridges (NTB), and takes advantage of address translated PCI or PCI Express write transactions. The *PXImc Software Specification* is designed to be used by both general purpose CPU networks and component-based instrument systems.

## 1.1 Objectives

The objectives of this specification are as follows:

- Define PXImc Software Requirements
- Provide a specification for data transfer between PCI(e) root complexes
    - CPU process to CPU process (physical memory to physical memory)
    - CPU process to PCI(e) device
    - PCI(e) device to CPU process
    - PCI(e) device to PCI(e) device
- Enable the highest possible performance of the underlying hardware
    - Highest bandwidth
    - Lowest latency
- Model API specification after existing API specs (such as PXI-6)
- Define a ‘thin’ API that supports data transfer and event generation
    - Provide sufficient functionality to support future high-level APIs
- API should be easy to spec, easy to implement, and easy to use
    - Suitable for the average device customer
- Maximize interoperability for users
    - Maximize the portability and interoperability of applications to multiple PXImc Logic Block Vendors’ implementations beyond the benefits of implementing the standardized PXImc API.
        - An application using a PXImc interface should be able to be developed in such a way that any PXImc Logic Block vendor could supply the interface and the application would continue to function without modification/recompilation.
    - Multiple PXImc Logic Block vendors should be able to co-exist on any system, and the coexistence should be transparent to client applications
        - An application using PXImc should be able to be developed in such a way that the application can be unaware of the number of PXImc Logic Block vendors on the system, and be able to easily interact with any or all PXImc Logic Blocks, regardless of vendor.
- Define environment agnostic interfaces and requirements so that this spec can be implemented in a variety of environments
    - C-based API
    - Easy for users to interact with from any programming language
    - Use simple data types
- Allow the format of the data transferred over PXImc to be determined by an upper-layer protocol (such as a 488.2 or raw data transfer protocol)
- Allow multiple client applications on a system to transfer data in parallel independently of each other

- Allow multiple simultaneous memory windows to be opened between PXImc Device and Primary System Host

The non-objectives of this specification are as follows:

- This version of the specification will not enable hot-plug
- This version of the specification will not enable “Virtual Mesh” (but “Virtual Mesh” support is expected to be added in some future version of this specification)
- This specification does not specify the format or interpretation of the data transferred over PXImc
- This specification does not specify any alternate methods for communicating across the PXImc other than using the PXImc API defined within this specification
- This specification does not specify the mechanism for PCI(e) device enumeration and PCI(e) device resource allocation
- This specification does not specify any specifics regarding PCI(e) tree topology

## 1.2 Intended Audience and Scope

This specification is primarily intended for product developers interested in implementing and leveraging software features of the PXImc technology. Hardware developers will be interested in using these software interfaces for identifying and describing the capabilities of PXImc hardware products. Likewise, software developers and systems integrators should take advantage of these software interfaces to manage PXImc resources. Additionally, product developers and systems integrators should reference the operating system framework definitions to ensure system-level interoperability. Note that the definitions and requirements described in this document apply to PXImc hardware components only (that is, hardware components defined by the *PXImc Hardware Specification*).

## 1.3 Background and Terminology

This section defines the acronyms and key words referred to throughout this specification. This specification uses the following acronyms and key words:

- **API**: Application Programming Interface.
- **Bandwidth**: The amount of data transmitted in a time unit.
- **BIOS**: Basic Input/Output System.
- **Client**: Sessions that will only be paired with remote server sessions. Refer to Section 3.3.2.1, *Opening a Session*.
- **Client Application**: Application that uses the PXImc API that is defined in Section 3.3, *API*.
- **Event**: An action initiated that can be received by some communication partner.
- **Interface**: An entity that logically connects one system to another system.
- **Latency**: The amount of time between an event or action being initiated by the transmitter and the event or action being received by the receiver.
- **Local Session**: A session that is opened on the local side of the interface.
- **Local Window**: From the perspective of the local session, the local window resides in physical memory that is present locally.
- **Logical**: A session where both the local window and the remote window addresses are provided by the PXImc Logic Block. Refer to Section 3.3.2.1, *Opening a Session*.
- **Non-Transparent Bridge**: Refer to the *PXImc Hardware Specification*.
- **OS**: Operating System.
- **Paired Session**: After a session has been created and the session pairing, as defined in Section 3.3.2.2, *Session Pairing*, has executed and found a compatible session on the remote side of the interface, the session that is paired with the local session is the "paired session".

- **PCI-Express Root Complex**: Refer to the *PXImc Hardware Specification*.
- **Peer**: Sessions that will only be paired with remote peer sessions. Refer to Section 3.3.2.1, *Opening a Session*.
- **Physical**: A session where either the local window or the remote window physical address is manually provided by the user. Refer to Section 3.3.2.1, *Opening a Session*.
- **Physical Memory**: Physically present units for data storage (RAM).
- **Primary System Host**: Refer to the *PXImc Hardware Specification*.
- **PXImc Device**: Refer to the *PXImc Hardware Specification*.
- **PXImc Logic Block**: Refer to the *PXImc Hardware Specification*.
- **PXImc Logic Block Vendor**: The vendor that implements the PXImc Logic Block.
- **Remote Session**: A session that is opened on the remote side of the interface.
- **Remote Window**: From the perspective of the local session, the remote window resides in physical memory that is present remotely. Accesses to the remote window result in an access across the interface.
- **Server**: Sessions that will only be paired with remote client sessions. Refer to Section 3.3.2.1, *Opening a Session*.
- **Session**: The value that represents a window request over a PXImc interface. Refer to Section 3.3.2, *Sessions*.
- **Shared Component**: The PXImc Dispatcher. Refer to Section 4. *PXImc Shared Component: PXImc Dispatcher*.
- **Virtual Mesh**: The ability to directly connect between two PXImc Devices using the PXImc API. Refer to Section 6. *Virtual Mesh*.

This specification uses several key words, which are defined as follows:

**RULE**: Rules SHALL be followed to ensure compatibility. A rule is characterized by the use of the words SHALL and SHALL NOT.

**RECOMMENDATION**: Recommendations consist of advice to implementers that will affect the usability of the final module. A recommendation is characterized by the use of the words SHOULD and SHOULD NOT.

**PERMISSION**: Permissions clarify the areas of the specification that are not specifically prohibited. Permissions reassure the reader that a certain approach is acceptable and will cause no problems. A permission is characterized by the use of the word MAY.

**OBSERVATION**: Observations spell out implications of rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules, so that the reader understands why the rule must be followed.

**MAY**: A key word indicating flexibility of choice with no implied preference. This word is usually associated with a permission.

**SHALL**: A key word indicating a mandatory requirement. Designers SHALL implement such mandatory requirements to ensure interchangeability and to claim conformance with the specification. This word is usually associated with a rule.

**SHOULD**: A key word indicating flexibility of choice with a strongly preferred implementation. This word is usually associated with a recommendation.

## 1.4 Applicable Documents

- *PCI Express External Cabling Specification,* Revision 1.0
- *PCI Express Base Specification,* Revision 1.1
- *PXI Express Hardware Specification,* Revision 1.0

- *PXImc Hardware Specification,* Draft 1
- *PCI Local Bus Specification,* Revision 2.3
- *PCI-to-PCI Bridge Architecture Specification,* Revision 1.2

# 2. PXImc Software Architecture Overview

## 2.1 Overview

The PXImc Hardware Specification defines the PCI(e) topology present between a Primary System Host and a PXImc Device.

**OBSERVATION**: The Non-Transparent Bridge appears as a PCI(e) device to the Primary System Host and also appears as a PCI(e) device to the PXImc Device.

Software for the PXImc hardware that resides within the Primary System Host is not addressed in this specification. This hardware can be viewed as an extension of the Primary System Host's PCI(e) tree, and should be transparent to the PXImc software.

**OBSERVATION**: PXImc hardware within the Primary System Host is completely interoperable with any vendor's PXImc Device hardware, assuming the two pieces of hardware use the same interconnect (cable & connector)

From the perspective of software, a single PXImc connection can be logically represented by Figure 2-1.

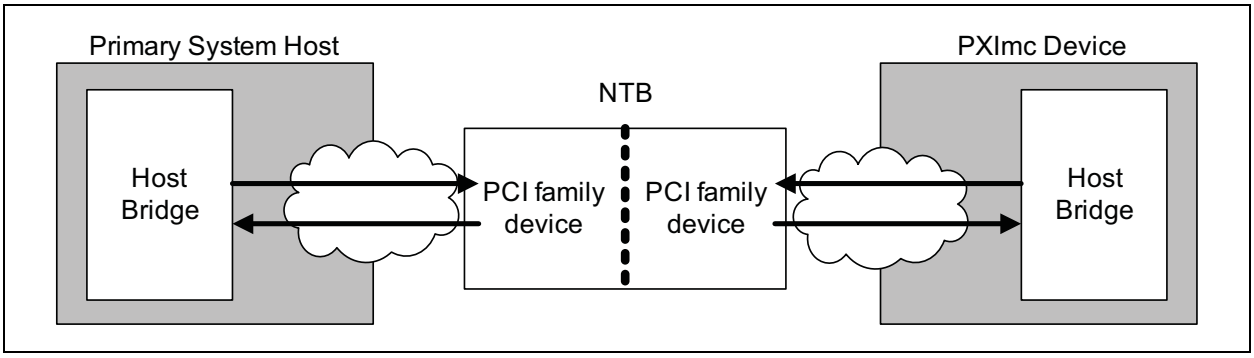


**Figure 2-1.** Software Visible Logical Connection

**OBSERVATION**: To software, the following details of the physical topology are all abstracted:
- Physical location of the NTB hardware
- Physical boundaries of the Primary System Host and the PXImc Device
- Interconnect method (cable or backplane)
- Bus technology (PCI or PCI-Express)

Software will be needed to run on both the Primary System Host and the PXImc Device to interact with the PXImc hardware. The vendor of the PXImc hardware (The PXImc Logic Block Vendor) will provide the software for device interaction on both the Primary System Host and the PXImc Device. Figure 2-2 indicates the software stacks on the Primary System Host and the PXImc Device. The dashed lines indicate the interfaces defined by this specification.

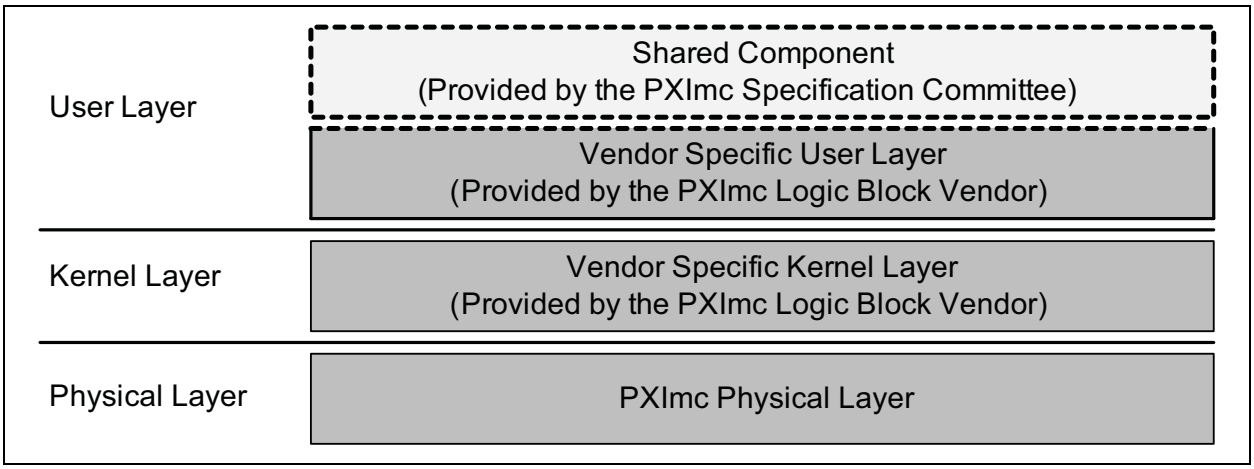


**Figure 2-2.** PXImc Software Model

The PXImc Software Specification defines an interface (the PXImc user mode API) that allows for data transactions/event generation. It does not define how this interface must be implemented; the implementation detail is left for the PXImc Logic Block Vendor. It does define many of the behaviors that must be provided by the interface, but allows the PXImc Logic Block Vendor implementation freedom, as long as the behaviors of the PXImc user mode API comply to the specified behaviors. This approach allows the PXImc Logic Block Vendor flexibility and freedom in many aspects of implementation (such as hardware selection).

## 2.2 PXImc Physical Layer

The Physical Layer provides the main functions required to setup and communicate with the physical hardware. This consists of the physical hardware and PCI(e) enumeration.

PCI(e) enumeration is typically provided by the host operating system and/or the system BIOS. Different operating systems will implement this functionality resulting in slightly different behavior. Typically, the OS will give an interface to a device driver developer to find and discover the hardware.

## 2.3 Vendor Specific Kernel Layer

The Kernel Layer behaviors are not defined by this specification. It is assumed that the Kernel Layer will help deliver the features needed to comply with the API specification. The Kernel Layer component will follow operating system specific conventions for associating with the PXImc hardware. While not required, it is assumed the Kernel Layer performs accesses to the PXImc hardware. The Kernel Layer may interact with the PXImc hardware in any way it chooses; its interaction with the hardware is not included in this specification. The Kernel Layer only exists on dual mode operating systems.

## 2.4 Vendor Specific User Layer

The minimum User Layer interface is defined by this specification. The User Layer SHALL export the API specification defined in Section 3.3, *API*. It is assumed that the User Layer will help deliver the features needed to comply with the API specification. The User Layer may have additional features or functions exported from it beyond the requirements of the API specification. The User Layer may interact with the Kernel Layer in any way it chooses; its interaction with the Kernel Layer is not included in this specification.

## 2.5 Shared Component Layer

A PXImc Shared Component is included in this specification so that components above the PXImc layer can load the shared component and make API calls that can be run regardless of the vendor of the PXImc. It is covered in detail in Section 4., *PXImc Shared Component: PXImc Dispatcher*.

# 3. API

## 3.1 Overview

This section defines the API that shall be implemented for the PXImc system.

## 3.2 Objectives

The following objectives were identified for the PXImc API

- Allow multiple simultaneous memory windows to be opened between two systems
- Enable vendor-independent interoperability of user applications
- Implement the PXImc API in the most environment-independent way possible
- Do not impede the performance of the hardware or add unnecessary overhead
- Allow for the API to be implemented on 32 or 64 bit environments
    - Do not preclude implementations where one side of the connection is a 32-bit environment and the other side of the connection is a 64-bit environment
- Allow for the API to be implemented on little-endian or big-endian environments
    - Do not preclude implementations where one side of the connection is a little-endian environment and the other side of the connection is a big-endian environment

The following non-objectives were identified for the PXImc API

- This API specification does not specify the format of the data being transferred, and instead allows protocols to run on top of PXImc to define the data format

## 3.3 API

A dynamic library implementing the functionality defined by this specification is called a PXImc Vendor-Specific User Layer.

**RULE**: A PXImc Vendor-Specific User Layer SHALL export all symbols by name.

**RULE**: A PXImc Vendor-Specific User Layer SHALL use the standard system calling convention for all entry points. On 32-bit Windows, this SHALL be stdcall.

**RULE**: A PXImc Vendor-Specific User Layer SHALL be thread safe, allowing multiple threads to simultaneously call functions.

**RULE**: A PXImc Vendor-Specific User Layer SHALL contain all of the API operations defined in Section 3.3. Each of the operations SHALL correspond to an exported symbol of the dynamic library.

All functions return a `tPXIMC_Status` value to indicate the status from the function. The value `PXIMC_SUCCESS` is used to indicate successful execution, negative numbers indicate an error condition, and positive numbers indicate a warning condition. All other output values are returned as arguments. The convention for output values is that the caller allocates the memory to contain the output value, and the API populates the caller-supplied memory with the value to return.

The API uses fundamental C data types to represent the data types given in the function definitions. All arguments are supplied and returned in the local native endianness.

**OBSERVATION**: The Vendor-Specific User Layer MAY need to translate endianness of some arguments so that all data provided to the local caller of the PXImc API is provided data in the local endianness.

## 3.3.1 Interfaces

### 3.3.1.1 PXIMC_findInterfaces

#### Purpose

Returns an array of unsigned integers that enumerate the available PXImc "interfaces" present in the system.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| maxNumberOfInterfaces | In | uint32_t | Size, in U32s, of the provided interfaceIDs buffer. |
| interfaceIDs | Out | uint32_t * | Array of interfaces. Each interface is represented by a single U32). |
| actualNumberOfInterfaces | Out | uint32_t * | The actual number of interfaces populated into the interfaceIDs array, or the number of interfaces present if insufficient space was provided in the interfaceIDs array. |

#### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The list of interfaces was successfully populated into the interfaceIDs parameter. |

| Error Codes | Description |
|---|---|
| PXIMC_INSUFFICIENT_SPACE | The maxNumberOfInterfaces parameter indicates that the space available in the interfaceIDs array isn't sufficient to hold the complete array of interfaces. |

| Warning Codes | Description |
|---|---|
| PXIMC_NO_PROVIDER | No vendor-specific user layer is registered with the PXImc Dispatcher. |

#### Description

The caller of PXIMC_findInterfaces allocates an array of U32s, and passes the address of the array and the size of the array as parameters. PXIMC_findInterfaces locates the interfaces present in the system that comply with the *PXImc Specification*, and returns the list of interfaces through the interfaceIDs parameter. The caller can then take additional action on one or more interface.

actualNumberOfInterfaces holds the number of interfaces written to the interfaceIDs array. If insufficient space is allocated by the user to hold the list of interfaces, actualNumberOfInterfaces is set to the actual number of interfaces present in the system.

#### Implementation Requirements

**RULE**: PXIMC_findInterfaces SHALL return at least one interface for every PXImc Logic Block where the logic block vendor has successfully initialized the local interface.

**RULE**: PXIMC_findInterfaces SHALL return an interface regardless of whether the remote side of the interface has been initialized.

**OBSERVATION**: Physically-present PXImc interfaces may not be returned by PXIMC_findInterfaces. Some reasons this can occur are:

- The PXImc interface may not have a device driver installed
- The PXImc device driver may not be properly associated with the PXImc interface

**RULE**: Interface numbers SHALL be unique.

**RULE**: Interface numbers SHALL NOT be zero.

**RECOMMENDATION**: A given interfaceID SHOULD be constant across power cycles and reboots for any specific PXImc Logic Block instance. If the hardware configuration across reboots is constant, then the interfaceIDs available SHOULD also be constant.

**RULE**: IF the maxNumberOfInterfaces argument is insufficient to hold the entire list of interfaces, the actualNumberOfInterfaces parameter SHALL be updated to contain the number of interfaces present. In this case the return value SHALL be PXIMC_INSUFFICIENT_SPACE.

**RULE**: IF the value passed in the maxNumberOfInterfaces argument is equal to or greater than the minimum size needed to hold the entire list of interfaces, THEN the value of actualNumberOfInterfaces SHALL hold the number of interfaces written to the interfaceIDs array.

**OBSERVATION**: An interface returned from PXIMC_findInterfaces represents the system on the remote side of the PXImc connection.

**OBSERVATION**: PXIMC_findInterfaces returns the interfaces available at the time PXIMC_findInterfaces is called. Subsequent system changes may cause changes in the list of available interfaces. To detect these changes, a program should re-call PXIMC_findInterfaces.

**OBSERVATION**: Executing PXIMC_findInterfaces is likely the initial call into the PXImc API. Further action is taken by executing a specific action against a specific interface returned by PXIMC_findInterfaces.

**OBSERVATION**: It is possible that zero interfaces are found, and therefore no interfaces written to the interfaceIDs buffer, if no interfaces are present.

### 3.3.1.2 PXIMC_queryInterfaceInformation

#### Purpose

Allow attributes of an interface to be queried.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to query the attribute. |
| attributeID | In | uint32_t | Attribute to query. |
| maxSizeOfAttributeValue | In | uint32_t | Size, in bytes, of the attributeValue buffer. |
| attributeValue | Out | void * | Attribute value. |
| actualSizeOfAttributeValue | Out | uint32_t * | The actual number of bytes written to the attributeValue buffer, or the size of the attribute if insufficient space was provided in the attributeValue array. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The attribute value was successfully populated into the attributeValue parameter. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_NSUP_ATTRIBUTE | Attribute requested is not supported. |
| PXIMC_INSUFFICIENT_SPACE | The maxSizeOfAttributeValue parameter indicates that the space available in the attributeValue buffer isn't sufficient to hold the complete attribute value. |
| PXIMC_ALIGNMENT_ERROR | The attributeValue buffer does not meet the alignment criteria required by the type of attribute requested. |

## Attributes

### String Attributes

| Attribute Name | Description | Required or optional |
|---|---|---|
| PXIMC_STR_MANF_NAME | The manufacturer name of the PXImc Logic Block hardware | Optional |
| PXIMC_STR_MODEL_NAME | The model name of the PXImc Logic Block | Optional |
| PXIMC_STR_SERIAL_NAME | The serial number of the PXImc Logic Block | Optional |
| PXIMC_STR_LOG_DATA | Any log data recorded against the interface | Optional |
| PXIMC_STR_INTERFACE_NAME | The name of the interface | Optional |
| PXIMC_STR_REMOTE_OS | The operating system running on the remote system | Optional |

### Unsigned 32-bit Integer Attributes

| Attribute Name | Description | Potential Values | Required or optional |
|---|---|---|---|
| PXIMC_U32_PROTOCOL_VERSION | The PXImc protocol version that this interface is compliant with. | PXIMC_SPEC_VERSION For the 1.0 version of the spec, this value is 0x00010000. | Required |
| PXIMC_U32_MANF_ID | The vendor-ID of the PXImc Logic Block Vendor | Any | Required |
| PXIMC_U32_INTERFACE_STATE | Indicates whether the remote side of the interface has been initialized, and whether this interface can currently be used to establish a connection to the remote system | PXIMC_STATE_UP PXIMC_STATE_DOWN | Required |
| PXIMC_U32_INTERFACE_DEVICE_ID | The device-ID of the PCI(e) device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_VENDOR_ID | The vendor-ID of the PCI(e) device that is providing the interface | Any | Optional |

| Attribute Name | Description | Potential Values | Required or optional |
|---|---|---|---|
| PXIMC_U32_INTERFACE_SS_ID | The subsystem ID of the PCI(e) device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_SS_VENDOR_ID | The subsystem vendor ID of the PCI(e) device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_BUS | The PCI(e) bus number of the device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_DEV | The PCI(e) device number of the device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_FUNC | The PCI(e) function number of the device that is providing the interface | Any | Optional |
| PXIMC_U32_INTERFACE_LOCAL | Indicates whether the physical interface is located in the local system | PXIMC_LOCAL<br>PXIMC_REMOTE | Optional |
| PXIMC_U32_REMOTE_ENDIANNESS | Returns how the value “0x12345678” is represented on the remote system if examined as an array of bytes, from the lowest address to the highest address | Any. Little-endian systems would return “0x78563412”. Big-endian systems would return “0x12345678”. | Optional |
| PXIMC_U32_REMOTE_WORD_SIZE | Returns the actual number of bits for the remote's native word size, e.g. the values 32 or 64 | Any. 64-bit systems would return ‘64’. 32-bit systems would return ‘32’ | Optional |

## Description

PXIMC_queryInterfaceInformation can be used to query the state of an attribute of the specified interface. The type of the return value should be interpreted based on the type of attribute queried.

actualSizeOfAttributeValue holds the number of bytes written to the attributeValue buffer. If insufficient space is allocated by the user to hold the attribute value, actualSizeOfAttributeValue is set to the actual number of bytes needed to hold the attribute value requested.

## Implementation Requirements

**RULE**: IF the value passed in the maxSizeOfAttributeValue argument is equal to or greater than the minimum size needed to hold the entire attribute value, THEN the value of actualSizeOfAttributeValue SHALL be set to the number of bytes written to the attributeValue buffer.

**RULE**: IF the value passed in the maxSizeOfAttributeValue argument is less than the minimum size needed to hold the entire attribute value, THEN the value of actualSizeOfAttributeValue SHALL be set to the minimum number of bytes needed to hold the attribute requested. In this case the return value SHALL be PXIMC_INSUFFICIENT_SPACE.

**RULE**: All attributes listed as Required MUST be implemented. These attributes SHALL NOT return PXIMC_NSUP_ATTRIBUTE.

**OBSERVATION**: All attributes listed as Optional MAY return PXIMC_NSUP_ATTRIBUTE.

**RULE**: The attribute IDs from 0xF0000000 – 0xFFFFFFFF, inclusive, are reserved for vendor-specific attributes. Vendor-specific attributes SHALL ONLY be implemented within the designated range.

## Unsigned 32-bit Integer Attributes

**RULE**: All unsigned 32-bit integer attribute values returned through PXIMC_queryInterfaceInformation SHALL be represented as native 32-bit unsigned integer.

**OBSERVATION**: The caller of PXIMC_queryInterfaceInformation can cast the attributeValue to an unsigned 32-bit integer and use the integer directly from the attributeValue buffer.

**OBSERVATION**: The exact size needed to hold an unsigned 32-bit integer attribute value is four.

**OBSERVATION**: IF the maxSizeOfAttributeValue is less than four, the size of a 32-bit Unsigned Integer, THEN PXIMC_queryInterfaceInformation SHALL NOT write any bytes to the attributeValue output buffer.

**RULE**: For unsigned 32-bit integer attributes, PXIMC_queryInterfaceInformation SHALL NOT write any bytes beyond the first four bytes of the attributeValue output buffer.

**RULE**: For unsigned 32-bit integer attributes, the attributeValue buffer SHALL be aligned on a 32-bit boundary.

**RULE**: PCI vendor IDs are defined to be 16-bits. PXIMC_U32_MANF_ID SHALL return zero in the upper two bytes of the attribute.

**RULE**: The attribute PXIMC_U32_INTERFACE_STATE SHALL ONLY return PXIMC_STATE_UP if all of the following conditions are satisfied:

- The local interface is initialized and has no errors
- The remote side of the interface is initialized and has no errors
- The local side of the interface is capable of communicating with the remote side of the interface
- The remote side of the interface is capable of communicating with the local side of the interface

In all other cases, PXIMC_U32_INTERFACE_STATE SHALL return PXIMC_STATE_DOWN.

The interpretation of the PXIMC_U32_PROTOCOL_VERSION attribute is that the upper two bytes indicate the major version number, and the lower two bytes indicate the minor version number.

### String Attributes

**RULE**: All string attribute values returned through `PXIMC_queryInterfaceInformation` SHALL be terminated by a NULL (‘\0’) character.

**RULE**: The NULL (‘\0’) character counts in the total size of the attribute. The `actualSizeOfAttributeValue` value SHALL include the NULL character.

**RULE**: All string attribute values SHALL ONLY contain bytes between 0x20 and 0x7E. Bytes 0x00 – 0x1F, and bytes 0x7F – 0xFF SHALL NOT be part of the `attributeValue` output.

**RULE**: The terminating NULL (‘\0’) character is required on string attributes. If an attribute value exactly fits into the `attributeValue` output buffer, but there is no space for the terminating NULL, the `attributeValue` SHALL NOT be returned. The NULL SHALL be considered as part of the `attributeValue` for computing size requirements.

## 3.3.1.3 PXIMC_waitForInterfaceEvent

### Purpose

Wait for an event to occur on an interface.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| `interfaceID` | In | `uint32_t` | Interface on which to wait for the event. |
| `timeoutInMilliseconds` | In | `uint32_t` | The amount of time to block waiting for an event to occur on the interface. |
| `reasonCode` | Out | `uint32_t *` | A bitmap of the specific event(s) that caused `PXIMC_waitForInterfaceEvent` to return. |

### Return Values

| Completion Code | Description |
|---|---|
| `PXIMC_SUCCESS` | An event has occurred on the given interface. |

| Error Codes | Description |
|---|---|
| `PXIMC_INVALID_INTERFACE` | The interface specified by `interfaceID` is not a valid interface. |

| Warning Codes | Description |
|---|---|
| `PXIMC_TIMEOUT` | The `timeoutInMilliseconds` duration elapsed without an event occurring. |

`PXIMC_waitForInterfaceEvent` is a blocking call waiting for an event to occur on the interface. `PXIMC_waitForInterfaceEvent` returns after either:

- an event listed in the `reasonCode` table occurs on the interface or
- the `timeoutInMilliseconds` duration elapses,

whichever occurs first.

timeoutInMilliseconds can be set to PXIMC_TIMEOUT_INFINITE to specify that the function should never return due to a timeout.

If an event does occur on the interface, the reasonCode indicates what specific event(s) occurred. The reasonCode value returned is a bitmap of events, with each bit representing a unique event. If multiple bits are set in reasonCode, multiple events have occurred. Table 3-1 lists the valid values for the reasonCode return value.

**Table 3-1.** reasonCode Return Values

| Value | Description |
|---|---|
| PXIMC_EVENT_INTERFACE_STATE_CHANGE | The state of the interface has changed. Use PXIMC_queryInterfaceInformation with attributeID = PXIMC_U32_INTERFACE_STATE to determine the current state of the interface. |
| PXIMC_EVENT_WINDOW_STATE_CHANGE | The available windows on the remote system have changed. Use PXIMC_findWindows to determine the current status of all remote windows. |

### Implementation Requirements

**RULE**: For every event type, the event queue on a session SHALL be implemented as a one-deep queue of events, with no notification of overflow of the event queue.

**OBSERVATION**: If multiple events of a specific event type occur while a process is not waiting for interface events, the next time the process waits for interface events it will be notified one time that an event occurred.

**OBSERVATION**: If a process receives a non-zero returnCode, the process should interpret the returnCode as one or more of the specified events has occurred since the last call to PXIMC_waitForInterfaceEvent.

**RULE**: reasonCode SHALL NOT be written unless PXIMC_waitForSessionEvent returns PXIMC_SUCCESS.

**RULE**: Notification of interface events is per process. Every process SHALL be treated independently regarding notification of events.

**OBSERVATION**: The first time any process calls PXIMC_waitForInterfaceEvent, it will return immediately.

**RULE**: Interface events SHALL be queued on behalf of every client application. The calling process need not be waiting on an interface event while an event occurs to be notified of an event next time it calls PXIMC_waitForInterfaceEvent.

**OBSERVATION**: When a process calls PXIMC_waitForInterfaceEvent it will be notified of any events that occurred on the interface since the last time the same process called PXIMC_waitForInterfaceEvent.

**OBSERVATION**: Callers of PXIMC_waitForInterfaceEvent should only call this function from one thread per process for a given interface. Calling this function from multiple threads will result in only one of the threads being notified of any specific event.

## 3.3.1.4 PXIMC_findWindows

### Purpose

Returns an array of unsigned integers that enumerate the PXImc "windows" present on the remote side of the interface.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to query the available windows. |
| maxNumberOfWindowIDs | In | uint32_t | Size, in U32s, of the provided windowIDs buffer. |
| windowIDs | Out | uint32_t * | Array of windows. Each window is represented by a single U32. |
| actualNumberOfWindowIDs | Out | uint32_t * | The actual number of windows populated into the windowIDs buffer, or the number of windows present if insufficient space was provided in the windowIDs array. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The windows were successfully populated into the windowIDs parameter. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified in interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INSUFFICIENT_SPACE | The maxNumberOfWindowIDs parameter indicates that the space available in the windowIDs buffer isn't sufficient to hold the complete list of windows. |

### Description

The caller of PXIMC_findWindows allocates an array of U32s, and passes the address of the array and the size of the array as parameters. PXIMC_findWindows locates the windows present on the remote system interface, and returns the list of windows through the windowIDs buffer. The caller can then take additional action on one or more window, such as calling PXIMC_queryWindowInformation, to determine specific attributes of each window. This data can be primarily used by "clients" and "peers" to determine information on what "server" and "peer" windows exist on the remote system, what the specific characteristics are of those "server" and "peer" sessions, and then determine whether it desires to connect to one of them as a "client" or "peer".

actualNumberOfWindowIDs holds the number of windows written to the windowIDs array. If insufficient space is allocated by the user to hold the complete list of windows, actualNumberOfWindowIDs is set to the actual number of U32 elements needed to hold the list of windows.

### Implementation Requirements

**RULE**: PXIMC_findWindows SHALL return one window ID for every window on the remote system represented by interface.

**OBSERVATION**: IF there are no sessions open on the remote system, THEN no window ID values will be present in the windowIDs.

**RULE**: Every window ID returned SHALL be the `uniqueIdentifier` of the window.

**OBSERVATION**: The window ID can be passed as a parameter during the window request as the `uniqueIdentifier` parameter to request a connection to a specific window on the remote system.

**OBSERVATION**: All `windowID` values for a given interface are guaranteed to be unique.

**RULE**: If the remote session was created with a `uniqueIdentifier` of zero, the `windowID` value returned SHALL be the unique identifier assigned to the session, and NOT zero.

**RULE**: IF an unpaired open session gets closed by `PXIMC_closeWindow`, THEN it shall no longer appear as a window ID value in the `windowIDs` on the remote system.

**OBSERVATION**: It is possible that an instance of a window ID that was returned by `PXIMC_findWindows` is no longer available when attempting to perform future accesses to it. This situation could exist if the window instance was either closed or paired between the call to `PXIMC_findWindows` and the future window request.

**RULE**: IF the `maxNumberOfWindowIDs` argument is insufficient to hold the entire list of windows the `actualNumberOfWindowIDs` parameter SHALL be updated to contain the number of windows present. In this case the return value SHALL be `PXIMC_INSUFFICIENT_SPACE`.

**RULE**: IF the value passed in the `maxNumberOfWindowIDs` argument is equal to or greater than the minimum size needed to hold the entire list of windows, THEN the value of `actualNumberOfWindowIDs` SHALL hold the number of windows written to the `windowIDs` array.

### 3.3.1.5 PXIMC_queryWindowInformation

#### Purpose

Allow attributes of a window to be queried.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| `interfaceID` | In | `uint32_t` | Interface on which to query a window attribute. |
| `windowID` | In | `uint32_t` | The specific window of which the attribute is being queried. |
| `attributeID` | In | `uint32_t` | Attribute to query. |
| `maxSizeOfAttributeValue` | In | `uint32_t` | Size, in bytes, of the `attributeValue` buffer. |
| `attributeValue` | Out | `void *` | The attribute value. |
| `actualSizeOfAttributeValue` | Out | `uint32_t *` | The actual number of bytes written to the `attributeValue` buffer, or the size of the attribute if insufficient space was provided in the `attributeValue` buffer. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The requested attribute value was successfully populated into the attributeID parameter. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified in interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_NSUP_ATTRIBUTE | Attribute requested is not supported. |
| PXIMC_INVALID_WINDOW | The window specified in windowID is not a valid window on the specified interface. |
| PXIMC_INSUFFICIENT_SPACE | The maxSizeOfAttributeValue parameter indicates that the space available in the attributeValue buffer isn't sufficient to hold the complete attribute value. |
| PXIMC_ALIGNMENT_ERROR | The attributeValue buffer does not meet the alignment criteria required by the type of attribute requested. |

### Attributes

### Unsigned 8-bit Array Attributes

| Attribute Name | Description | Required or optional |
|---|---|---|
| PXIMC_U8_WINDOW_DATA | The window data array of bytes supplied when the remote window was created. | Required |

### Unsigned 32-bit Integer Attributes

| Attribute Name | Description | Valid Return Values | Required or optional |
|---|---|---|---|
| PXIMC_U32_WINDOW_ CONNECTION_TYPE | The connection type used to create the remote window. | PXIMC_CONNECTION_SERVER<br>PXIMC_CONNECTION_CLIENT<br>PXIMC_CONNECTION_PEER | Required |
| PXIMC_U32_WINDOW_ LOCATION_TYPE | The location type used to create the remote window. | PXIMC_LOCATION_LOGICAL<br>PXIMC_LOCATION_PHYSICAL | Required |
| PXIMC_U32_WINDOW_ PROTOCOL_NUMBER | The protocol number supplied when the remote window was created. | 0x0 – 0xFFFFFFFF | Required |

| Attribute Name | Description | Valid Return Values | Required or optional |
|---|---|---|---|
| PXIMC_U32_WINDOW_PAIRING_STATE | Indicates whether the remote window has previously been paired with a local window, or if it is currently unpaired and available for pairing. | PXIMC_WINDOW_PAIRED<br>PXIMC_WINDOW_UNPAIRED | Required |
| PXIMC_U32_SESSION_EVENT_STATUS | On paired windows, determine: 1) If the local session has an outstanding event 2) If the local session is currently waiting on an event, 3) If the remote session has an outstanding event and 4) If the remote session is currently waiting on an event. | 0x0 – 0x0000000F<br>(0 – PXIMC_WINDOW_REMOTE_EVENT_PENDING \| PXIMC_WINDOW_REMOTE_SESSION_WAITING \| PXIMC_WINDOW_LOCAL_EVENT_PENDING \| PXIMC_WINDOW_LOCAL_SESSION_WAITING) | Optional |

### Unsigned 64-bit Integer Attributes

| Attribute Name | Description | Valid Return Values | Required or optional |
|---|---|---|---|
| PXIMC_U64_WINDOW_MIN_REMOTE_SIZE | The minimum remote window size supplied when the remote window was created. | 0x0 – 0xFFFFFFFFFFFFFFFF | Required |
| PXIMC_U64_WINDOW_MAX_REMOTE_SIZE | The maximum remote windowsize supplied when the remote window was created. | 0x0 – 0xFFFFFFFFFFFFFFFF | Required |
| PXIMC_U64_WINDOW_MIN_LOCAL_SIZE | The minimum local window size supplied when the remote window was created. | 0x0 – 0xFFFFFFFFFFFFFFFF | Required |
| PXIMC_U64_WINDOW_MAX_LOCAL_SIZE | The maximum local window size supplied when the remote window was created. | 0x0 – 0xFFFFFFFFFFFFFFFF | Required |

## Description

PXIMC_queryWindowInformation can be used to query an attribute of a requested window on the specified interface. The type of the return value should be interpreted based on the type of attribute queried.

The specific attribute values of the windows open on the remote system can be useful to potential local “clients” and “peers” to determine information on specific attributes of “server” and “peer” windows that exist on the remote system. The local system can use these attribute values to determine whether it desires to connect to one of the remote windows as a “client” or “peer”.

actualSizeOfAttributeValue holds the number of bytes written to the attributeValue buffer. If insufficient space is allocated by the user to hold the attribute value, actualSizeOfAttributeValue is set to the actual number of bytes needed to hold the attribute value requested.

**OBSERVATION**: If using PXIMC_queryWindowInformation to find information on available windows to connect with, the PXIMC_U32_WINDOW_PAIRING_STATE must be queried to determine if the window is currently unpaired.

### Implementation Requirements

**RULE**: The attributes of every window SHALL match the parameter values provided to the window request when the session was created on the remote system.

**OBSERVATION**: The PXIMC_WINDOW_MIN_REMOTE_SIZE, PXIMC_WINDOW_MAX_REMOTE_SIZE, PXIMC_WINDOW_MIN_LOCAL_SIZE, and PXIMC_WINDOW_MAX_LOCAL_SIZE attribute values are the values requested by the remote session. To connect to a given remote session, the local and remote window sizes must be swapped for the local window request.

**RULE**: IF the value passed in the maxSizeOfAttributeValue argument is equal to or greater than the minimum size needed to hold the entire attribute value, THEN the value of actualSizeOfAttributeValue SHALL be set to the number of bytes written to the attributeValue buffer.

**RULE**: IF the value passed in the maxSizeOfAttributeValue argument is less than the minimum size needed to hold the entire attribute value, THEN the value of actualSizeOfAttributeValue SHALL be set to the minimum number of bytes needed to hold the attribute value requested. In this case the return value SHALL be PXIMC_INSUFFICIENT_SPACE.

### Unsigned 8-bit Array Attributes

**RULE**: The attributeValue provided by the caller must be an allocated buffer of the size maxSizeOfAttributeValue.

**RULE**: PXIMC_queryWindowInformation SHALL NOT write any more bytes in attributeValue than is indicated by actualSizeOfAttributeValue.

### Unsigned 32-bit Integer Attributes

**RULE**: All unsigned 32-bit integer attribute values returned through PXIMC_queryWindowInformation SHALL be represented as native 32-bit unsigned integer.

**OBSERVATION**: The caller of PXIMC_queryWindowInformation can cast the attributeValue to an unsigned 32-bit integer and use the integer directly from the attributeValue buffer.

**OBSERVATION**: The exact size needed to hold an unsigned 32-bit integer attribute value is four.

**RULE**: For unsigned 32-bit integer attributes, PXIMC_queryWindowInformation SHALL NOT write any bytes beyond the first four bytes of the attributeValue output buffer.

**RULE**: For unsigned 32-bit integer attributes, the attributeValue buffer SHALL be aligned on a 32-bit boundary.

### Unsigned 64-bit Integer Attributes

**RULE**: All unsigned 64-bit integer attribute values returned through PXIMC_queryWindowInformation SHALL be represented as native 64-bit unsigned integer.

**OBSERVATION**: The caller of PXIMC_queryWindowInformation can cast the attributeValue to an unsigned 64-bit integer and use the integer directly from the attributeValue buffer.

**OBSERVATION**: The exact size needed to hold an unsigned 64-bit integer attribute value is eight.

**RULE**: For unsigned 64-bit integer attributes, `PXIMC_queryWindowInformation` SHALL NOT write any bytes beyond the first eight bytes of the `attributeValue` output buffer.

**RULE**: For unsigned 64-bit integer attributes, the `attributeValue` buffer SHALL be aligned on an 8-byte boundary.

**RULE**: For paired windows, the value of `PXIMC_U64_WINDOW_MIN_REMOTE_SIZE` SHALL be equal to the value of `PXIMC_U64_WINDOW_MAX_REMOTE_SIZE`. Both values SHALL be the actual size of the remote window.

**RULE**: For paired windows, the value of `PXIMC_U64_WINDOW_MIN_LOCAL_SIZE` SHALL be equal to the value of `PXIMC_U64_WINDOW_MAX_LOCAL_SIZE`. Both values SHALL be the actual size of the local window.

## 3.3.2 Sessions

### 3.3.2.1 Opening a Session

A session is opened by requesting a window against a specific interface. There are five functions that allow a window to be requested. Each function is a combination of a window location type (either a physical or logical window location) and a window connection type (either server, client, or peer).

Throughout this document there are several references to requesting a window. When this document refers to requesting a window, it is an indication that it is generically referencing the set of functions that allow a window to be requested. There are five specific functions that allow a window to be requested (`PXIMC_requestWindowLogicalAsServer`, `PXIMC_requestWindowLogicalAsClient`, `PXIMC_requestWindowLogicalAsPeer`, `PXIMC_requestWindowPhysicalAsServer`, `PXIMC_requestWindowPhysicalAsClient`).

The following tables give an introduction to the differences between the different functions for requesting a window.

#### Physical vs. Logical:

| Window location type | Description |
|---|---|
| `Logical` | Requests a “logical” connection over the PXImc interface. A logical connection enables a process on the local system to communicate with a process on the remote system. |
| `Physical` | Requests a “physical” connection over the PXImc interface. A physical connection either:<br>• Enables the local process to write to a physical entity (such as a hardware device) on the remote system.<br>This operation is requesting a remote window. A remote window must be requested. A client window connection type must be used for this type of window.<br>• Enables a process on the remote system to write to a physical entity (such as a hardware device) on the local system.<br>This operation is requesting a local window. A local window must be requested. A `physicalAddress` must be specified in this request (the `physicalAddress` parameter must be non-zero). A server window connection type must be used for this type of window. |

### Server vs. Client vs. Peer:

| Window connection type | Description |
|---|---|
| `Client` | Follow the client rules for session initialization and session pairing. The client rules are:<br>Clients only will be paired with server sessions.<br>If a server on the remote system isn't available for pairing at the time the client requests a window, the window request will fail. |
| `Server` | Follow the server rules for session initialization and session pairing. The server rules are:<br>Servers will only be paired with client sessions.<br>If a client on the remote system isn't available for pairing at the time the server requests its window, the window request will succeed. The server window request will be broadcast to the remote system, and the remote system can become aware of the server window request by calling `PXIMC_findWindows`. The properties of the server request will be broadcast to the remote system and be queryable using `PXIMC_queryWindowInformation`. Potential clients on the remote system can use `PXIMC_findWindows` and `PXIMC_queryWindowInformation` to be aware of the available server session and receive the server attributes. The server is notified of session pairing by `PXIMC_waitForConnection`. |
| `Peer` | Follow the peer rules for session initialization and session pairing. The peer rules are:<br>Peers will only be paired with peer sessions.<br>If a peer on the remote system isn't available for pairing at the time the local peer requests its window, the window request will succeed. The peer window request will be broadcast to the remote system, and the remote system can become aware of the peer window request by calling `PXIMC_findWindows`. The properties of the peer request will be broadcast to the remote system and be queryable using `PXIMC_queryWindowInformation`. Potential peers on the remote system can use `PXIMC_findWindows` and `PXIMC_queryWindowInformation` to be aware of the available peer session and receive the peer attributes. The peer is notified of session pairing by `PXIMC_waitForConnection`. |

The following apply to all of the functions that allow a window to be requested:

**RULE**: A new session can only be opened by requesting a window.

**OBSERVATION**: No communication can occur over the interface immediately after a window request has completed. Requesting a window is the first step in initiating communication across the interface.

**RULE**: If a session is successfully created by requesting a window, the session cannot be used from any process other than the one that created it.

**RULE**: The `sessionNumber` argument SHALL NOT be written to UNLESS the return value is `PXIMC_SUCCESS`.

**RULE**: The `sessionNumber` returned SHALL be unique.

**RULE**: The `sessionNumber` returned SHALL NOT be zero.

**OBSERVATION**: `sessionNumbers` can be reused only after the previous session has been closed.

### 3.3.2.2 Session Pairing

I/O is not possible over a new session after only requesting a window. The session must first be "paired" with a session created on the system represented by the `interfaceID` argument to the window request. The remote session that is paired with the local session is referred to as the "paired session". All of the pairing criteria must be fulfilled for two sessions to be paired.

**OBSERVATION**: Session Pairing may occur during the window request.

**RULE**: If two sessions are not precluded from being paired by the following criteria, then the two sessions SHALL be paired together:

- Window Location Type:
  - Logical will only pair with other logical. Logical SHALL NOT pair with physical.
  - Physical will only pair with other physical. Physical SHALL NOT pair with logical.
- Window Connection Type:
  - Peers will only pair with other peers. Peers SHALL NOT pair with servers or clients.
  - Clients will only pair with servers. Clients SHALL NOT pair with clients or peers.
  - Servers will only pair with clients. Servers SHALL NOT pair with servers or peers.
- Requested Window Sizes:
  - There must be overlap in the requested window sizes. For the local window, take the maximum value of `minLocalSize` and the remote session's `minRemoteSize`. This is the net minimum window size. Take the minimum value of the local `maxLocalSize` and the remote session's `maxRemoteSize`. This is the net maximum window size. If the net maximum window size is less than the net minimum window size, there is no overlap in requested window sizes. The same operation must be performed on the local session's remote window and the remote session's local window. Sessions SHALL NOT be paired if either of the windows does not have overlap in requested window sizes.
  - The two net maximum window sizes must not be zero. If both net maximum window sizes are zero, the two sessions SHALL NOT be paired.
- Resource Availability:
  - The two net minimum window sizes must both be available to be satisfied by the resources available to the PXImc interface. If both net minimum window sizes cannot be satisfied, the two sessions SHALL NOT be paired.
  - Some resources MUST be reserved for the connection. If both net minimum window sizes are zero, some resources need to be allocated to one of the two windows, not to exceed the net maximum window size. If no resources are available for either window, the two sessions SHALL NOT be paired.
- `protocolNumber`: Both sessions must pass the same value for `protocolNumber`. Sessions with different `protocolNumbers` SHALL NOT be paired.
- `uniqueIdentifier`:
  - If both the remote session and the local session pass a non-zero value for the `uniqueIdentifier`, the value must match. Non-matching non-zero `uniqueIdentifiers` SHALL NOT be paired.
  - If connecting as `CONNECTION_PEER` or `CONNECTION_CLIENT` AND the session pairing algorithm is being evaluated during the window request AND the value of the `uniqueIdentifier` passed in is zero, THEN `uniqueIdentifier` is not used as a pairing criteria.

**OBSERVATION**: There are many timing pitfalls/race conditions that must be avoided to guarantee the pairing criteria are followed. These need to be accounted for in implementations of the PXImc Logic Block.

**OBSERVATION**: It is possible that the session pairing criteria yield more than one potential match. In this case, any one of the potential matches will actually be paired.

**OBSERVATION**: The `windowData` is never used in the session pairing criteria.

**OBSERVATION**: To pair a remote session with a local session, the local session's remote window size must be equal to the remote session's local window size, and the local session's local window size must be equal to the remote session's remote window size.

**OBSERVATION**: After a remote session is paired with a local session, the local session's remote window size will be less than or equal to the `maxRemoteSize` specified in the local window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's remote window size will be greater than or equal to the `minRemoteSize` specified in the local window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's remote window size will be less than or equal to the `maxLocalSize` specified in the remote window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's remote window size will be greater than or equal to the `minLocalSize` specified in the remote window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's local window size will be less than or equal to the `maxLocalSize` specified in the local window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's local window size will be greater than or equal to the `minLocalSize` specified in the local window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's local window size will be less than or equal to the `maxRemoteSize` specified in the remote window request.

**OBSERVATION**: After a remote session is paired with a local session, the local session's local window size will be greater than or equal to the `minRemoteSize` specified in the remote window request.

**RULE**: Sessions SHALL be paired with only one remote session. After a session has been paired with one remote session, it SHALL NOT be evaluated in the future for potential pairing.

**RULE**: The connection SHALL be allocated as close to the net maximum window sizes as can be serviced given the available resources to the PXImc interface.

**OBSERVATION**: Two sessions can be paired and be allocated any window sizes between the net maximum window size and the net minimum window size.

**RULE**: Once two sessions have been paired, the resources assigned to the connection SHALL be reserved exclusively for that connection. The resources consumed by the connection SHALL NOT be considered as available when evaluating future session pairing.

## 3.3.2.3 Request Window API

### 3.3.2.3.1 PXIMC_requestWindowLogicalAsServer

#### Purpose

Initiate a request for a memory window to communicate across the PXImc interface. A logical window is requested that follows the server rules for the request and session pairing.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to request the window. |
| protocolNumber | In | uint32_t | The protocol that will be used to communicate with the paired session. Refer to Section 5., *Protocols*. |
| maxLocalSize | In | uint64_t | The maximum size of the local window desired. |
| minLocalSize | In | uint64_t | The minimum size of the local window desired. |
| maxRemoteSize | In | uint64_t | The maximum size of the remote window desired. |
| minRemoteSize | In | uint64_t | The minimum size of the remote window desired. |
| uniqueIdentifier | In | uint32_t | A unique ID to uniquely describe this window. |
| windowData | In | const uint8_t * | A buffer to broadcast across the interface to describe this session. |
| windowDataSize | In | uint32_t | The size, in bytes, of the windowData byte array. |
| sessionNumber | Out | uint32_t * | The session number that corresponds to this request. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The window was successfully requested. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid, or a combination of the arguments provided is invalid. |
| PXIMC_SPACE_NOT_AVAILABLE | Either the minimum size of the local window, or the remote window, or both is not available on the interfaceID interface. |
| PXIMC_UID_CONFLICT | The uniqueIdentifier specified is already in use. |

### Description

A logical window is requested that will follow the server rules for initializing the connection. The session created by PXIMC_requestWindowLogicalAsServer will be used to communicate with a process on the remote system. The session will only be paired with client sessions on the remote system, and the local session will be visible to the remote system when the remote system calls PXIMC_findWindows.

## Memory Windows

PXIMC_requestWindowLogicalAsServer allows for two separate windows to be requested, a local window and a remote window. It is important to understand how these two memory windows operate to determine which window(s) to request.

The local window is a memory range that is backed by physical memory on the local system. The remote window is a memory range that is backed by physical memory on the remote system. After the local session is paired with a remote session, local accesses to the remote window are translated by the PXImc interface into an access on the remote system. For example, if the local session writes data to the remote window, the PXImc interface receives the write, and transparently writes that data to the remote system. After the local session is paired with a remote session, the local session's remote window directly maps to the remote session's local window. The two windows (the local session's remote window and the remote session's local window) are guaranteed to be the same size, and offsets within the two windows always correspond (data written to first byte of the local session's remote window will appear in the first byte of the remote session's local window). Reads or writes are possible from both the local window and the remote window. Full-duplex communication can be accomplished if only one of the two windows is used.

**RECOMENDATION**: For optimal performance, data to be sent to the paired session will be written to the remote window. Writing data to the remote window "pushes" the data to the remote system; the remote system can then read the data from its local window.

## Memory Window Sizes

Two parameters are provided for both the local and remote window sizes, a minimum size parameter and a maximum size parameter. The maximum size parameter specifies the maximum request for this session. The maximum value is a size that the window shall not exceed. The maximum value parameter can be set to PXIMC_MAXIMUM_WINDOW_SIZE to indicate a request to make the window as large as possible. The maximum value parameter can be set to zero to indicate the session will not accept a window of this kind. The maximum value parameter also indicates the preferred size of the window. The minimum size parameter specifies the minimum requirement to be allocated for this session to allow the connection to complete. The combination of the maximum and minimum parameter values help determine the actual size of the window. For example, if the maximum size parameter is set to PXIMC_MAXIMUM_WINDOW_SIZE and the minimum size parameter is zero, the remote session's window request completely determines the size of the window. If both the maximum and minimum size parameters are set to zero, it indicates that no window will be allocated.

For example, if the minimum size is set to 0x400 (1KB) and the maximum size is set to 0x1000 (4KB), this means that if 4KB is available that the window size should be 4KB, if less is available, down to 1KB, that the connection should still complete. If any less than 1KB is available, the connection should not complete, as less than 1KB is insufficient for this session to operate. The session is notified of the window size that is allocated to the connection after a successful session pairing has completed by calling PXIMC_waitForConnection.

## Unique Identifier

A zero value indicates that PXImc should automatically assign some unique identifier to this session; that the caller has no preference what unique identifier is used. A non-zero value manually assigns a specific unique identifier to this session. Regardless of whether the unique identifier was automatically or manually assigned, the unique identifier assigned to this session is posted as one of the windowIDs in the results of PXIMC_findWindows on the remote system. This uniqueIdentifier allows a client to uniquely target a specific server to initiate a connection with.

### Window Data

The window data is a way to pass data about the local session to the remote system. The data passed to the remote system is up to the caller of PXIMC_requestWindowLogicalAsServer. The data passed in through the windowData parameter is available to the remote system through PXIMC_findWindows and PXIMC_queryWindowInformation.

The intent of this argument is to provide data that the potential client or peer on the remote system can use to determine whether it wants to initiate a connection with this server/peer.

The data passed in this argument MUST not be longer than 1024 bytes. The windowDataSize tells how many bytes are present in the window data. It is not required to pass any data through the windowData parameter. If the use of the windowData is not desired, windowDataSize can be set to zero.

### Implementation Requirements

**RULE**: The following sequence indicates this API's behavior SHALL:

- Validate interfaceID is a valid interface. If not, return PXIMC_INVALID_INTERFACE.
- Validate that at least one window is being requested (at least one of the two max size parameters is non-zero). If both zero, return PXIMC_INVALID_ARGUMENT.
- Validate that the maxLocalSize parameter is greater than or equal to the minLocalSize parameter. If it is not, return PXIMC_INVALID_ARGUMENT.
- Validate that the maxRemoteSize parameter is greater than or equal to the minRemoteSize parameter. If it is not, return PXIMC_INVALID_ARGUMENT.
- Verify windowDataSize is less than or equal to 1024. If not, return PXIMC_INVALID_ARGUMENT.
- Determine if minimum window size(s) can be satisfied. Does the PXImc interface physically have sufficient resources to accommodate the minimum requirements of the PXIMC_requestWindowLogicalAsServer caller? If not, return PXIMC_SPACE_NOT_AVAILABLE.
- If the uniqueIdentifier parameter is non-zero, determine if any connection or resource on the local system against this interface is currently using the unique identifier provided. If yes, return PXIMC_UID_CONFLICT.
- If the uniqueIdentifier parameter is zero, determine a uniqueIdentifier value that is not currently in-use on the local system against this interface. Assign this uniqueIdentifier to this requested window.
- "Post" this requested window, and all its attributes (the protocolNumber, maxLocalSize, minLocalSize, maxRemoteSize, minRemoteSize, uniqueIdentifier, and windowData) to the remote system.
- Generate a unique session number that is not in-use for any PXImc interface. Set the sessionNumber parameter to this new session number.
- Return PXIMC_SUCCESS.

**RULE**: Either the maxLocalSize or the maxRemoteSize SHALL be allowed to be zero when requesting a window.

**OBSERVATION**: Initiating a connection where one of the two window sizes is zero allows the two connected systems to share a single segment of memory physically present in one of the two systems. Both systems can read or write the shared physical memory.

## 3.3.2.3.2 PXIMC_requestWindowLogicalAsClient

### Purpose

Initiate a request for a memory window to communicate across the PXImc interface. A logical window is requested that follows the client rules for the request and session pairing.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to request the window. |
| protocolNumber | In | uint32_t | The protocol that will be used to communicate with the paired session. Refer to Section 5., *Protocols*. |
| maxLocalSize | In | uint64_t | The maximum size of the local window desired. |
| minLocalSize | In | uint64_t | The minimum size of the local window desired. |
| maxRemoteSize | In | uint64_t | The maximum size of the remote window desired. |
| minRemoteSize | In | uint64_t | The minimum size of the remote window desired. |
| uniqueIdentifier | In | uint32_t | A unique ID to uniquely describe this window. |
| sessionNumber | Out | uint32_t * | The session number that corresponds to this request. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The window was successfully requested. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid, or a combination of the arguments provided is invalid. |
| PXIMC_SPACE_NOT_AVAILABLE | Either the minimum size of the local window, or the remote window, or both is not available on the interfaceID interface. |
| PXIMC_NO_PAIRING | The local request could not be paired with a compatible remote session. |

### Description

A logical window is requested that will follow the client rules for initializing the connection. The session created by PXIMC_requestWindowLogicalAsClient will be used to communicate with a process on the remote system. The session will only be paired with server sessions on the remote system, and the call to PXIMC_requestWindowLogicalAsClient will fail if a compatible server is not available on the remote system at the time PXIMC_requestWindowLogicalAsClient is called.

**OBSERVATION**: PXIMC_requestWindowLogicalAsClient can be used with PXIMC_findWindows and PXIMC_queryWindowInformation. PXIMC_findWindows and PXIMC_queryWindowInformation allows potential client applications to determine available server sessions on the remote system, and the client application can use the data from PXIMC_queryWindowInformation to determine the values to pass to PXIMC_requestWindowLogicalAsClient.

## Memory Windows

PXIMC_requestWindowLogicalAsClient allows for two separate windows to be requested, a local window and a remote window. It is important to understand how these two memory windows operate to determine which window(s) to request.

The local window is a memory range that is backed by physical memory on the local system. The remote window is a memory range that is backed by physical memory on the remote system. After the local session is paired with a remote session, local accesses to the remote window are translated by the PXImc interface into an access on the remote system. For example, if the local session writes data to the remote window, the PXImc interface receives the write, and transparently writes that data to the remote system. After the local session is paired with a remote session, the local session's remote window directly maps to the remote session's local window. The two windows (the local session's remote window and the remote session's local window) are guaranteed to be the same size, and offsets within the two windows always correspond (data written to first byte of the local session's remote window will appear in the first byte of the remote session's local window). Reads or writes are possible from both the local window and the remote window. Full-duplex communication can be accomplished if only one of the two windows is used.

**RECOMENDATION**: For optimal performance, data to be sent to the paired session will be written to the remote window. Writing data to the remote window "pushes" the data to the remote system; the remote system can then read the data from its local window.

## Memory Window Sizes

Two parameters are provided for both the local and remote window sizes, a minimum size parameter and a maximum size parameter. The maximum size parameter specifies the maximum request for this session. The maximum value is a size that the window shall not exceed. The maximum value parameter can be set to PXIMC_MAXIMUM_WINDOW_SIZE to indicate a request to make the window as large as possible. The maximum value parameter can be set to zero to indicate the session will not accept a window of this kind. The maximum value parameter also indicates the preferred size of the window. The minimum size parameter specifies the minimum requirement to be allocated for this session to allow the connection to complete. The combination of the maximum and minimum parameter values help determine the actual size of the window. For example, if the maximum size parameter is set to PXIMC_MAXIMUM_WINDOW_SIZE and the minimum size parameter is zero, the remote session's window request completely determines the size of the window. If both the maximum and minimum size parameters are set to zero, it indicates that no window will be allocated.

For example, if the minimum size is set to 0x400 (1KB) and the maximum size is set to 0x1000 (4KB), this means that if 4KB is available that the window size should be 4KB, if less is available, down to 1KB, that the connection should still complete. If any less than 1KB is available, the connection should not complete, as less than 1KB is insufficient for this session to operate. The session is notified of the window size that is allocated to the connection after a successful session pairing has completed by calling PXIMC_waitForConnection.

## Unique Identifier

A zero value indicates the client does not want the unique identifier to be used in the session pairing algorithm. A non-zero value indicates that the client will only connect with a server that has been assigned the unique identifier value passed into the function.

## Implementation Requirements

**RULE**: The following sequence indicates the behavior PXIMC_requestWindowLogicalAsClient SHALL provide:

- Validate interfaceID is a valid interface. If not, return PXIMC_INVALID_INTERFACE.
- Validate that at least one window is being requested (at least one of the two max size parameters is non-zero). If both zero, return PXIMC_INVALID_ARGUMENT.

- Validate that the maxLocalSize parameter is greater than or equal to the minLocalSize parameter. If it is not, return PXIMC_INVALID_ARGUMENT.
- Validate that the maxRemoteSize parameter is greater than or equal to the minRemoteSize parameter. If it is not, return PXIMC_INVALID_ARGUMENT.
- Determine if minimum window size(s) can be satisfied. Does the PXImc interface physically have sufficient resources to accommodate the minimum requirements of the PXIMC_requestWindowLogicalAsClient caller? If not, return PXIMC_SPACE_NOT_AVAILABLE.
- Run the session pairing algorithm defined in Section 3.3.2.2, *Session Pairing*. If the session pairing algorithm does not successfully locate a remote session to pair with this local session, return PXIMC_NO_PAIRING.
- Record the paired session information in internal data structures.
- Update internal available resources data structures to indicate the resources consumed by this connection are no longer available.
- Generate a unique session number that is not in-use for any PXImc interface. Set the sessionNumber parameter to this new session number.
- Return PXIMC_SUCCESS.

**RULE**: Either the maxLocalSize or the maxRemoteSize SHALL be allowed to be zero when requesting a window.

**OBSERVATION**: Initiating a connection where one of the two window sizes is zero allows the two connected systems to share a single segment of memory physically present in one of the two systems. Both systems can read or write the shared physical memory.

#### 3.3.2.3.3 PXIMC_requestWindowLogicalAsPeer

##### Purpose

Initiate a request for a memory window to communicate across the PXImc interface. A logical window is requested that follows the peer rules for the request and session pairing.

##### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to request the window. |
| protocolNumber | In | uint32_t | The protocol that will be used to communicate with the paired session. Refer to Section 5., *Protocols*. |
| maxLocalSize | In | uint64_t | The maximum size of the local window desired. |
| minLocalSize | In | uint64_t | The minimum size of the local window desired. |
| maxRemoteSize | In | uint64_t | The maximum size of the remote window desired. |
| minRemoteSize | In | uint64_t | The minimum size of the remote window desired. |
| uniqueIdentifier | In | uint32_t | A unique ID to uniquely describe this window. |
| windowData | In | const uint8_t * | A buffer to broadcast across the interface to describe this session. |
| windowDataSize | In | uint32_t | The size, in bytes, of the windowData byte array. |
| sessionNumber | Out | uint32_t * | The session number that corresponds to this request. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The window was successfully requested. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid, or a combination of the arguments provided is invalid. |
| PXIMC_SPACE_NOT_AVAILABLE | Either the minimum size of the local window, or the remote window, or both is not available on the interfaceID interface. |
| PXIMC_UID_CONFLICT | The uniqueIdentifier specified is already in use. |

### Description

A logical window is requested that will follow the peer rules for initializing the connection. The session created by PXIMC_requestWindowLogicalAsPeer will be used to communicate with a process on the remote system. The session will only be paired with peer sessions on the remote system, and the call to PXIMC_requestWindowLogicalAsPeer will first attempt to locate a compatible peer session on the remote system. If a compatible remote peer session is found, the local session will be paired with it. If a compatible remote peer session is not found, the local session will be visible to the remote system when the remote system calls PXIMC_findWindows.

**OBSERVATION**: A ‘peer’ session allows for either the remote session or the local session to be created first. Ordering of session creation is not important when using a ‘peer’ session.

### Memory Windows

PXIMC_requestWindowLogicalAsPeer allows for two separate windows to be requested, a local window and a remote window. It is important to understand how these two memory windows operate to determine which window(s) to request.

The local window is a memory range that is backed by physical memory on the local system. The remote window is a memory range that is backed by physical memory on the remote system. After the local session is paired with a remote session, local accesses to the remote window are translated by the PXImc interface into an access on the remote system. For example, if the local session writes data to the remote window, the PXImc interface receives the write, and transparently writes that data to the remote system. After the local session is paired with a remote session, the local session’s remote window directly maps to the remote session’s local window. The two windows (the local session’s remote window and the remote session’s local window) are guaranteed to be the same size, and offsets within the two windows always correspond (data written to first byte of the local session’s remote window will appear in the first byte of the remote session's local window). Reads or writes are possible from both the local window and the remote window. Full-duplex communication can be accomplished if only one of the two windows is used.

**RECOMENDATION**: For optimal performance, data to be sent to the paired session will be written to the remote window. Writing data to the remote window "pushes" the data to the remote system; the remote system can then read the data from its local window.

## Memory Window Sizes

Two parameters are provided for both the local and remote window sizes, a minimum size parameter and a maximum size parameter. The maximum size parameter specifies the maximum request for this session. The maximum value is a size that the window shall not exceed. The maximum value parameter can be set to `PXIMC_MAXIMUM_WINDOW_SIZE` to indicate a request to make the window as large as possible. The maximum value parameter can be set to zero to indicate the session will not accept a window of this kind. The maximum value parameter also indicates the preferred size of the window. The minimum size parameter specifies the minimum requirement to be allocated for this session to allow the connection to complete. The combination of the maximum and minimum parameter values help determine the actual size of the window. For example, if the maximum size parameter is set to `PXIMC_MAXIMUM_WINDOW_SIZE` and the minimum size parameter is zero, the remote session's window request completely determines the size of the window. If both the maximum and minimum size parameters are set to zero, it indicates that no window will be allocated.

For example, if the minimum size is set to 0x400 (1KB) and the maximum size is set to 0x1000 (4KB), this means that if 4KB is available that the window size should be 4KB, if less is available, down to 1KB, that the connection should still complete. If any less than 1KB is available, the connection should not complete, as less than 1KB is insufficient for this session to operate. The session is notified of the window size that is allocated to the connection after a successful session pairing has completed by calling `PXIMC_waitForConnection`.

## Unique Identifier

The peer connection first attempts to match against the currently available peers on the remote system. During the matching phase, the unique identifier argument has the same rules as defined in the 'Client' description. If the peer connection remains unpaired after the matching phase is complete, the value of the unique identifier then is interpreted with the 'Server' description below.

**Server**: A zero value indicates that PXImc should automatically assign some unique identifier to this session; that the caller has no preference what unique identifier is used. A non-zero value manually assigns a specific unique identifier to this session. Regardless of whether the unique identifier was automatically or manually assigned, the unique identifier assigned to this session is posted as one of the `windowIDs` in the results of `PXIMC_findWindows` on the remote system. This `uniqueIdentifier` allows a peer to uniquely target a specific peer to initiate a connection with.

**Client**: A zero value indicates the client does not want the unique identifier to be used in the session pairing algorithm. A non-zero value indicates that the client will only connect with a server that has been assigned the unique identifier value passed into the function.

## Window Data

The window data is a way to pass data about the local session to the remote system. The data passed to the remote system is up to the caller of `PXIMC_requestWindowLogicalAsServer`. The data passed in via the `windowData` parameter is available to the remote system through `PXIMC_findWindows` and `PXIMC_queryWindowInformation`.

The intent of this argument is to provide data that the potential client or peer on the remote system can use to determine whether it wants to initiate a connection with this server/peer.

The data passed in this argument MUST not be longer than 1024 bytes. The `windowDataSize` tells how many bytes are present in the window data. It is not required to pass any data through the `windowData` parameter. If the use of the `windowData` is not desired, `windowDataSize` can be set to zero.

## Implementation Requirements

**RULE**: The following sequence indicates the behavior `PXIMC_requestWindowLogicalAsPeer` SHALL provide:

- Validate `interfaceID` is a valid interface. If not, return `PXIMC_INVALID_INTERFACE`.
- Validate that at least one window is being requested (at least one of the two max size parameters is non-zero). If both zero, return `PXIMC_INVALID_ARGUMENT`.
- Validate that the `maxLocalSize` parameter is greater than or equal to the `minLocalSize` parameter. If it is not, return `PXIMC_INVALID_ARGUMENT`.
- Validate that the `maxRemoteSize` parameter is greater than or equal to the `minRemoteSize` parameter. If it is not, return `PXIMC_INVALID_ARGUMENT`.
- Verify `windowDataSize` is less than or equal to 1024. If not, return `PXIMC_INVALID_ARGUMENT`.
- Determine if minimum window size(s) can be satisfied. Does the PXImc interface physically have sufficient resources to accommodate the minimum requirements of the `PXIMC_requestWindowLogicalAsPeer` caller. If not, return `PXIMC_SPACE_NOT_AVAILABLE`.
- Run the session pairing algorithm defined in Section 3.3.2.2, *Session Pairing*.
- Branch on whether the session pairing algorithm successfully locates a remote session to pair with this local session.

  IF PAIR FOUND:

  - Record the paired session information in internal data structures.
  - Update internal available resources data structures to indicate resources consumed by this connection are no longer available.
  - Generate a unique session number that is not in-use for any PXImc interface. Set the `sessionNumber` parameter to this new session number.
  - Return `PXIMC_SUCCESS`.

  IF NO PAIR FOUND:

  - If the `uniqueIdentifier` parameter is non-zero, determine if any connection or resource on the local system against this interface is currently using the unique identifier provided. If yes, return `PXIMC_UID_CONFLICT`.
  - If the `uniqueIdentifier` parameter is zero, determine a `uniqueIdentifier` value that is not currently in-use on the local system against this interface. Assign this `uniqueIdentifier` to this requested window.
  - "Post" this requested window, and all its attributes (the `protocolNumber`, `maxLocalSize`, `minLocalSize`, `maxRemoteSize`, `minRemoteSize`, and `windowData`) to the remote system.
  - Generate a unique session number that is not in-use for any PXImc interface controlled by this PXImc Logic Block Vendor. Set the `sessionNumber` parameter to this new session number.
  - Return `PXIMC_SUCCESS`.

**RULE**: Either the `maxLocalSize` or the `maxRemoteSize` SHALL be allowed to be zero when requesting a window.

**OBSERVATION**: Initiating a connection where one of the two window sizes is zero allows the two connected systems to share a single segment of memory physically present in one of the two systems. Both systems can read or write the shared physical memory.

#### 3.3.2.3.4 PXIMC_requestWindowPhysicalAsServer

### Purpose

Initiate a request for a memory window to communicate across the PXImc interface. A physical window is requested that follows the server rules for the request and session pairing.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| interfaceID | In | uint32_t | Interface on which to request the window. |
| protocolNumber | In | uint32_t | The protocol that will be used to communicate with the paired session. Refer to Section 5., *Protocols*. |
| localSize | In | uint64_t | The size of the local window desired. |
| uniqueIdentifier | In | uint32_t | A unique ID to uniquely describe this window. |
| physicalAddress | In | uint64_t | The local physical address to enable as a target for a remote session. |
| windowData | In | const uint8_t * | A buffer to broadcast across the interface to describe this session. |
| windowDataSize | In | uint32_t | The size, in bytes, of the windowData byte array. |
| sessionNumber | Out | uint32_t * | The session number that corresponds to this request. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The window was successfully requested. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid, or a combination of the arguments provided is invalid. |
| PXIMC_SPACE_NOT_AVAILABLE | Either the minimum size of the local window, or the remote window, or both is not available on the interfaceID interface. |
| PXIMC_UID_CONFLICT | The uniqueIdentifier specified is already in use. |
| PXIMC_PHY_RESOURCE_NOT_AVAILABLE | The physical resource requested is not available, or cannot fulfill the minimum needs of the requested window. |

### Description

A physical window is requested that will follow the server rules for initializing the connection. The session created by PXIMC_requestWindowPhysicalAsServer will be used to allow remote sessions to communicate with specific physical resource on the local system, such as a hardware device. The session will only be paired with client sessions on the remote system, and the local session will be visible to the remote system when the remote system calls PXIMC_findWindows.

**OBSERVATION**: PXIMC_requestWindowPhysicalAsServer can not be used to communicate with the remote system. It enables the remote system to target a specific range of physical address space on the local system.

## Memory Windows

PXIMC_requestWindowPhysicalAsServer allows a local window to be requested. It is important to understand how this memory window operates to determine how to use it. The local window is a memory range that is backed by some physical entity, such as physical memory, on the local system. The location of the local window in physical address space is specified by the physicalAddress parameter. After the local session is paired with a remote session, when the remote session accesses its remote window, those accesses are translated by the PXImc interface into an access on the local system. After the local session is paired with a remote session, the remote session's remote window directly maps to the local session's requested local window. The two windows (the local session's local window and the remote session's remote window) are guaranteed to be the same size, and offsets within the two windows always correspond (data written to first byte of the local remote window will appear in the first byte of the remote session's local window).

## Local Window Size

The localSize parameter specifies the exact size of the local resource that the remote session will be able to target directly. Once this session is paired with a remote session, the remote session will be able to target the following range in local physical address space: [physicalAddress through physicalAddress + localSize].

## Physical Address

The physical address parameter is used to allow the local session to enable the remote system to directly access a specific location within the local physical address space.

The most common scenario where the local session wants to enable direct access to a specific physical address is if the remote system will be allowed to directly read from and write to a specific piece of hardware within the local system. PXImc is capable of allowing a remote system to directly access local hardware present in physical address space, such as PCI and PCIe devices. To enable the remote system to make direct access to hardware through the PXImc interface, the base address of the physical address region for the hardware should be provided in the physicalAddress parameter.

## Unique Identifier

A zero value indicates that PXImc should automatically assign some unique identifier to this session; that the caller has no preference what unique identifier is used. A non-zero value manually assigns a specific unique identifier to this session. If a specific physical resource is desired to be used for this window request, its resourceID returned from PXIMC_findPhysicalResources should be supplied for the unique identifier parameter. If a unique identifier is specified that doesn't correspond with physical resource ID, the user has no preference which physical resource is used to request the window. Regardless of whether the unique identifier was automatically or manually assigned, the unique identifier assigned to this session is posted as one of the windowIDs in the results of PXIMC_findWindows on the remote system. This uniqueIdentifier allows a client to uniquely target a specific server to initiate a connection with.

## Window Data

The window data is a way to pass data about the local session to the remote system. The data passed to the remote system is up to the caller of PXIMC_requestWindowLogicalAsServer. The data passed in through the windowData parameter is available to the remote system through PXIMC_findWindows and PXIMC_queryWindowInformation.

The intent of this argument is to provide data that the potential client or peer on the remote system can use to determine whether it wants to initiate a connection with this server/peer.

The data passed in this argument MUST not be longer than 1024 bytes. The windowDataSize tells how many bytes are present in the window data. It is not required to pass any data through the windowData parameter. If the use of the windowData is not desired, windowDataSize can be set to zero.

### Implementation Requirements

**RULE**: The following sequence indicates the behavior `PXIMC_requestWindowPhysicalAsServer` SHALL provide:

- Validate `interfaceID` is a valid interface. If not, return `PXIMC_INVALID_INTERFACE`.
- Verify `windowDataSize` is less than or equal to 1024. If not, return `PXIMC_INVALID_ARGUMENT`.
- Validate the `localSize` is nonzero. If it is zero, return `PXIMC_INVALID_ARGUMENT`.
- Verify the `physicalAddress` parameter is non-zero. If not, return `PXIMC_INVALID_ARGUMENT`.
- Determine if the window size can be satisfied, and if the PXImc interface has the physical capabilities to satisfy the request. Does the PXImc interface physically have sufficient resources to accommodate the minimum requirements of the caller? If not, return `PXIMC_SPACE_NOT_AVAILABLE`.
- If the `uniqueIdentifier` parameter is non-zero, determine if any connection on the local system against this interface is currently using the unique identifier provided. If yes, return `PXIMC_UID_CONFLICT`.
- If the `uniqueIdentifier` parameter is zero, determine a `uniqueIdentifier` value that is not currently in-use on the local system against this interface. Assign this `uniqueIdentifier` to this requested window.
- "Post" this requested window, and all its attributes (the `protocolNumber`, `localSize`, `physicalAddress`, `uniqueIdentifier`, and `windowData`) to the remote system.
- Generate a unique session number that is not in-use for any PXImc interface. Set the `sessionNumber` parameter to this new session number.
- Return `PXIMC_SUCCESS`.

## 3.3.2.3.5 PXIMC_requestWindowPhysicalAsClient

### Purpose

Initiate a request for a memory window to communicate across the PXImc interface. A physical window is requested that follows the client rules for the request and session pairing.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| `interfaceID` | In | `uint32_t` | Interface on which to request the window. |
| `protocolNumber` | In | `uint32_t` | The protocol that will be used to communicate with the paired session. Refer to Section 5., *Protocols*. |
| `maxRemoteSize` | In | `uint64_t` | The maximum size of the remote window desired. |
| `minRemoteSize` | In | `uint64_t` | The minimum size of the remote window desired. |
| `uniqueIdentifier` | In | `uint32_t` | A unique ID to uniquely describe this window. |
| `sessionNumber` | Out | `uint32_t *` | The session number that corresponds to this request. |

### Return Values

| Completion Code | Description |
|---|---|
| `PXIMC_SUCCESS` | The window was successfully requested. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_INTERFACE | The interface specified by interfaceID is not a valid interface. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid, or a combination of the arguments provided is invalid. |
| PXIMC_SPACE_NOT_AVAILABLE | Either the minimum size of the local window, or the remote window, or both is not available on the interfaceID interface. |
| PXIMC_UID_CONFLICT | The uniqueIdentifier specified is already in use. |
| PXIMC_NO_PAIRING | The local request could not be paired with a compatible remote session. |
| PXIMC_PHY_RESOURCE_NOT_AVAILABLE | The physical resource requested is not available, or cannot fulfill the minimum needs of the requested window. |

## Description

A physical window is requested that will follow the client rules for initializing the connection. The session created by PXIMC_requestWindowPhysicalAsClient will be used to communicate with a physical entity, such as a hardware device, on the remote system. The session will only be paired with server sessions on the remote system, and the call to PXIMC_requestWindowPhysicalAsClient will fail if a compatible server is not available on the remote system at the time PXIMC_requestWindowPhysicalAsClient is called.

**OBSERVATION**: PXIMC_requestWindowPhysicalAsClient can be used with PXIMC_findWindows and PXIMC_queryWindowInformation. PXIMC_findWindows and PXIMC_queryWindowInformation allows potential client applications to determine available server sessions on the remote system, and the client application can use the data from PXIMC_queryWindowInformation to determine the values to pass to PXIMC_requestWindowLogicalAsClient.

**OBSERVATION**: PXIMC_requestWindowPhysicalAsClient has both a maxRemoteSize and a minRemoteSize because callers of PXIMC_requestWindowPhysicalAsClient may be compatible with multiple different types of callers of PXIMC_requestWindowPhysicalAsServer. By allowing the callers of PXIMC_requestWindowPhysicalAsClient to specify a size range, it allows the PXIMC_requestWindowPhysicalAsServer caller to specify the exact size of the window.

## Memory Windows

PXIMC_requestWindowPhysicalAsClient allows a remote window to be requested. It is important to understand how this memory window operates to determine how to use it. The remote window is a memory range that is backed by some physical entity, such as physical memory, on the remote system. The location of the remote window in physical address space on the remote system depends on the paired sessions value it supplied for the physicalAddress parameter. After the local session is paired with a remote session, when the local session accesses its local remote window, those accesses are translated by the PXImc interface into an access on the remote system, directly to an offset from the physical address provided by the remote system.

### Memory Window Sizes

Two parameters are provided for the remote window size, a minimum size parameter and a maximum size parameter. The maximum value is a size that the window shall not exceed. The minimum size parameter specifies the minimum requirement to be allocated for this session to allow the connection to complete. The call to `PXIMC_requestWindowPhysicalAsClient` can only succeed if the maximum size parameter is greater than or equal to the exact window size specified by the server. Also, the call to `PXIMC_requestWindowPhysicalAsClient` can only succeed if the minimum size parameter is less than or equal to the exact window size specified by the server. Regardless of the values supplied for both the maximum and minimum parameter values, the only possible size of the remote window is the specific size requested by the server. The session is notified of the window size that is allocated to the connection after a successful session pairing has completed by calling `PXIMC_waitForConnection`.

### Unique Identifier

A zero value indicates the client does not want the unique identifier to be used in the session pairing algorithm. A non-zero value indicates that the client will only connect with a server that has been assigned the unique identifier value passed into the function.

### Implementation Requirements

**RULE**: The following sequence indicates the behavior `PXIMC_requestWindowPhysicalAsClient` SHALL provide:

- Validate `interfaceID` is a valid interface. If not, return `PXIMC_INVALID_INTERFACE`.
- Validate that both `minRemoteSize` and `maxRemoteSize` are non-zero. If either are zero, return `PXIMC_INVALID_ARGUMENT`.
- Validate that the `maxRemoteSize` parameter is greater than or equal to the minRemoteSize parameter. If it is not, return `PXIMC_INVALID_ARGUMENT`.
- Determine if the minimum window size can be satisfied, and if the PXImc interface has the physical capabilities to satisfy the request. Does the PXImc interface physically have sufficient resources to accommodate the minimum requirements of the caller? If not, return `PXIMC_SPACE_NOT_AVAILABLE`.
- Run the session pairing algorithm defined in Section 3.3.2.2, *Session Pairing*. If the session pairing algorithm does not successfully locate a remote session to pair with this local session, return `PXIMC_NO_PAIRING`.
- Record the paired session information in internal data structures.
- Move the physical resource being used out of the available pool in internal data structures, and into the consumed pool.
- Generate a unique session number that is not in-use for any PXImc interface. Set the `sessionNumber` parameter to this new session number.
- Return `PXIMC_SUCCESS`.

## 3.3.2.4 Session Pairing API

### 3.3.2.4.1 PXIMC_waitForConnection

### Purpose

Determine if the local session has been successfully paired with a remote session. If a pairing has been established, return the information necessary to communicate with the remote session.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session on which to wait for a connection. |
| timeoutInMilliseconds | In | uint32_t | The amount of time, in milliseconds, to block waiting for a connection. |
| mappedRemoteAddress | Out | void ** | The mapped remote address that the process can use to access the remote window. |
| remoteSizeInBytes | Out | uint64_t * | The size of the remote window. |
| mappedLocalAddress | Out | void ** | The mapped remote address that the process can use to access the local window. |
| localSizeInBytes | Out | uint64_t * | The size of the local window. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The session has been successfully paired with a remote session. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_SESSION_CLOSED | The session was paired and then the remote session closed its session. |

| Warning Codes | Description |
|---|---|
| PXIMC_TIMEOUT | The timeoutInMilliseconds duration elapsed without the session successfully pairing. |

### Description

PXIMC_waitForConnection is a blocking call waiting for the local session to be paired with a remote session using the session pairing rules described in Section 3.3.2.2, *Session Pairing*.
PXIMC_waitForConnection returns after either

- the local session is successfully paired with a remote session or
- the timeoutInMilliseconds duration elapses,

whichever occurs first.

timeoutInMilliseconds can be set to PXIMC_TIMEOUT_INFINITE to specify that the function should never return due to a timeout.

If the local session is successfully paired with a remote session, the remoteSizeInBytes and the localSizeInBytes indicate the size of the remote and local windows, and the mappedRemoteAddress and mappedLocalAddress indicate the base address of the remote and local windows.

The mappedRemoteAddress and mappedLocalAddress values can only be used from the process that requested the window and called PXIMC_waitForConnection.

**OBSERVATION**: The mappedRemoteAddress and mappedLocalAddress can now be read and/or written to communicate with the remote session. The rules for how these two addresses are used are defined by the protocolNumber that was used to initiate the connection.

### Implementation Requirements

**RULE**: The four output parameters (`mappedLocalAddress`, `mappedRemoteAddress`, `localSizeInBytes`, and `remoteSizeInBytes`) SHALL NOT be modified unless `PXIMC_waitForConnection` returns `PXIMC_SUCCESS`.

**RULE**: `remoteSizeInBytes` SHALL NOT be greater than the net maximum remote window size, as defined in Section 3.3.2.2, *Session Pairing*.

**RULE**: `remoteSizeInBytes` SHALL NOT be less than the net minimum remote window size, as defined in Section 3.3.2.2, *Session Pairing*.

**RULE**: `localSizeInBytes` SHALL NOT be greater than the net maximum local window size, as defined in Section 3.3.2.2, *Session Pairing*.

**RULE**: `localSizeInBytes` SHALL NOT be less than the net minimum local window size, as defined in Section 3.3.2.2, *Session Pairing*.

**RULE**: `PXIMC_waitForConnection` SHALL NOT block for `timeoutInMilliseconds` if an error has occurred. It SHALL return immediately.

**RULE**: IF the interface goes down while a caller is blocked in `PXIMC_waitForConnection`, `PXIMC_waitForConnection` SHALL return as soon as possible after the interface goes down, and SHALL NOT wait until the timeout elapses.

**RULE**: IF the session was opened using `PXIMC_requestWindowPhysicalAsServer`, both the `mappedRemoteAddress` and `mappedLocalAddress` values SHALL be returned as NULL. Although the session was successfully paired, the local session in this case can not perform accesses to either window. The local window initialized for this session is directed at the `physicalAddress`, not at physical memory.

**RULE**: IF the session was opened using `PXIMC_requestWindowPhysicalAsClient`, the `mappedLocalAddress` SHALL be returned as NULL and the `localSizeInBytes` SHALL be zero, as no local window can be requested with `PXIMC_requestWindowPhysicalAsClient`.

**RULE**: IF the session has no local window, the `localSizeInBytes` SHALL be zero, and the `mappedLocalAddress` SHALL be NULL.

**RULE**: IF the session has no remote window, the `remoteSizeInBytes` SHALL be zero, and the `mappedRemoteAddress` SHALL be NULL.

**OBSERVATION**: The two values `mappedLocalAddress` and `mappedRemoteAddress` are addresses that the process can interact with directly. The addresses must be mapped to the processes address space.

**OBSERVATION**: The `mappedLocalAddress` and `mappedRemoteAddress` cannot be distributed to other processes for use.

**OBSERVATION**: A value of zero can be passed to `timeoutInMilliseconds` to poll on whether the local session has already been paired with a remote session.

**OBSERVATION**: Communication can occur between the local session and the paired session after `PXIMC_waitForConnection` has successfully completed.

**OBSERVATION**: The only way `PXIMC_waitForConnection` can return with `PXIMC_SUCCESS` is if the session has been paired with a remote session using the Session Pairing rules in Section 3.3.2.2, *Session Pairing*.

**OBSERVATION**: If the `maxLocalSize` specified in the corresponding window request is zero, the `mappedLocalAddress` and the `localSizeInBytes` arguments may be NULL. Similarly, if the `maxRemoteSize` specified in the corresponding window request is zero, the `mappedRemoteAddress` and the `remoteSizeInBytes` arguments may be NULL.

## 3.3.3 Window Physical Addresses

### 3.3.3.1 PXIMC_getPhysicalAddress

#### Purpose

Retrieve the raw physical address of the remote window for the session.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session on which to wait for a connection. |
| physicalAddress | Out | uint64_t * | The physical address that corresponds to the remote window for the given sessionNumber. |

#### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The physical address has successfully been written to the physicalAddress pointer. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_NO_PAIRING | The session specified by sessionNumber is an open session, but the session is not paired with a remote session. |
| PXIMC_NO_WINDOW | No remote window was allocated for the session. |
| PXIMC_SESSION_CLOSED | The session was paired and then the remote session closed its session. |

#### Description

PXIMC_getPhysicalAddress allows users to query the physical address of the session's remote window. This can be used to allow hardware or a kernel driver to directly source data over the connection. The session must have previously been paired and have opened a remote window to use this function.

**OBSERVATION**: If direct physical address accesses are used to source data over the connection, the mappedRemoteAddress parameter returned from PXIMC_waitForConnection should not be used. If both are used, it's likely that one data source will overwrite the other data source, potentially resulting in corrupted data that is incomprehensible by the remote session.

**OBSERVATION**: The user of this function must take special care to not exceed the physical address returned in remoteSizeInBytes. If data is written beyond the allocated size of remote window one or both systems may become unstable and the results of accesses outside the remote window are undefined.

#### Implementation Requirements

**RULE**: If the PXImc Logic Block Vendor has implemented the remote window in some way where the physical remote window is not a continuous region of physical address space, this function SHALL return an error.

### 3.3.3.2 PXIMC_enableDeviceAccess

#### Purpose

Enable a device to directly access the remote window physical address that corresponds to a given session.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session on which to enable device access. |
| accessMode | In | uint32_t | Mode the device will use to access the remote window. |
| deviceBusNumber | In | uint32_t | The PCI or PCI Express bus number of the device that will perform the access. |
| deviceDevNumber | In | uint32_t | The PCI or PCI Express device number of the device that will perform the access. |
| deviceFuncNumber | In | uint32_t | The PCI or PCI Express function number of the device that will perform the access. |

#### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The specified device has been enabled to read from the session. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_NO_PAIRING | The session specified by sessionNumber is an open session, but the session is not paired with a remote session. |
| PXIMC_NO_WINDOW | No remote window was allocated for the session. |
| PXIMC_SESSION_CLOSED | The session was paired and then the remote session closed its session. |
| PXIMC_PHY_RESOURCE_NOT_AVAILABLE | Insufficient physical resources are available to satisfy the request. |
| PXIMC_INVALID_ARGUMENT | One or more of the parameters provided is invalid. |

## Description

PXIMC_enableDeviceAccess will allow a PCI or PCI Express device to directly perform reads or writes of the remote window of the specified sessionNumber. The session must have been previously paired and have opened a remote window to use this function. Pass in one or more of the following values in the accessMode parameter.

| Value | Description |
|---|---|
| PXIMC_DEVICE_ACCESS_READ | The specified device desires read access to the remote window. |
| PXIMC_DEVICE_ACCESS_WRITE | The specified device desires write access to the remote window. |
| PXIMC_DEVICE_ACCESS_CLEAR_ALL | The specified device will have all of its previously enabled device accesses cleared. |

**RULE:** accessMode values are allowed to be 'OR'ed together. For example, PXIMC_DEVICE_ACCESS_READ | PXIMC_DEVICE_ACCESS_WRITE is a valid value to pass into the accessMode parameter.

**RULE:** PXIMC_DEVICE_ACCESS_CLEAR_ALL is a special accessMode, and should not be 'OR'ed with any other value.

**RULE:** This function must be called before attempting to access the physical address of the remote window.

**OBSERVATION:** Some systems do not support PCI or PCI Express devices directly reading or writing to another PCI or PCI Express device. Other systems support this functionality, but only in a subset of possible system configurations. This function does not address or work around those system limitations. Check with your system vendor to ensure this functionality is supported.

**OBSERVATION:** Typically, after successful return of PXIMC_enableDeviceAccess, the caller will call PXIMC_getPhysicalAddress to get the physical address to access. The remote window physical address must be obtained before the window can be addressed.

**OBSERVATION:** Every call to PXIMC_enableDeviceAccess is device specific. This function must be called once for every accessor.

**OBSERVATION:** By default, all devices in the system have no access, and should not perform reads or writes without requesting access using PXIMC_enableDeviceAccess. Behavior is undefined if a device initiates a transaction before PXIMC_enableDeviceReads has been called, or if the device initiates a transaction type that wasn't requested in accessMode. Therefore, PXIMC_DEVICE_ACCESS_CLEAR_ALL needs to be used only to clear previously granted access, and is not needed as part of standard initialization.

## Implementation Requirements

**RULE:** If any undefined/reserved bits in accessMode are set, or if any of deviceBusNumber, deviceDevNumber, or deviceFuncNumber are out of range of the legal values defined by the PCI specification, PXIMC_INVALID_ARGUMENT should be returned.

**OBSERVATION:** No validation that the requested deviceBusNumber, deviceDevNumber, and deviceFuncNumber apply to a present device will be performed.

**OBSERVATION:** If a session has been enabled for device access, the device access should get implicitly cleared when the session is closed.

## 3.3.4 Session Events

### 3.3.4.1 PXIMC_assertEvent

#### Purpose

Assert an event to the paired session.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session on which to assert the event. |

#### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | An event has successfully been asserted to the remote session. |

| Error Codes | Description |
|---|---|
| PXIMC_INTERFACE_DOWN | The interface specified in interfaceID cannot currently access the system on the remote side of the interface. |
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |
| PXIMC_NO_PAIRING | The session specified by sessionNumber is an open session, but the session is not paired with a remote session. |
| PXIMC_SESSION_CLOSED | The session was paired and then the remote session closed its session. |

#### Description

PXIMC_assertEvent allows for a session to trigger an PXIMC_EVENT_ASSERTED event to the remote session. The remote session will be notified of receiving the PXIMC_EVENT_ASSERTED if it is currently blocked in PXIMC_waitForSessionEvent, or next time it calls PXIMC_waitForSessionEvent.

PXIMC_assertEvent can be used to notify the remote session of a specific event, such as that the local session has completed sending data to it. The rules for when an event is asserted should be defined by the protocolNumber that was used to initiate the connection.

**OBSERVATION**: PXIMC_assertEvent does not block waiting for the remote session to receive the event.

**OBSERVATION**: Events function as if there is an event flag that can either be set or cleared. When the local session calls PXIMC_assertEvent, it sets the remote session's event flag. The local session can call PXIMC_assertEvent as many times as it wants, but the flag stays set once it is in the set state. Whenever the remote session calls, and returns from PXIMC_waitForSessionEvent, the flag is cleared. If a caller calls PXIMC_waitForSessionEvent when the flag is cleared, PXIMC_waitForSessionEvent blocks until the flag is in the set state.

#### Implementation Requirements

**RULE**: It SHALL be guaranteed that any data written to the remote window by the local session WILL be present in the local window of the remote session by the time the PXIMC_EVENT_ASSERTED is propagated to the remote session.

**OBSERVATION**: Some NTB implementations may not be able to guarantee the above rule, and the mechanism for guaranteeing the above rule may vary depending on the specific NTB implementation selected.

**RECOMMENDATION**: As event latency is a primary concern for users of PXImc, the event SHOULD be delivered to the remote session efficiently.

### 3.3.4.2 PXIMC_waitForSessionEvent

#### Purpose

Wait to receive an event from the remote session.

#### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session on which to wait for a session event. |
| timeoutInMilliseconds | In | uint32_t | The amount of time to block waiting for the paired session to cause an event. |
| reasonCode | Out | uint32_t * | The specific event that caused PXIMC_waitForSessionEvent to return. |

#### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | An event has occurred on the given session. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |
| PXIMC_NO_PAIRING | The session specified by sessionNumber is an open session, but the session is not paired with a remote session. |

| Warning Codes | Description |
|---|---|
| PXIMC_TIMEOUT | The timeoutInMilliseconds duration elapsed without an event occurring. |

#### Description

PXIMC_waitForSessionEvent is a blocking call waiting for the remote session to generate some event. PXIMC_waitForSessionEvent returns after either

- the remote session generates an event listed in the reasonCode table or
- the timeoutInMilliseconds duration elapses,

whichever occurs first.

timeoutInMilliseconds can be set to PXIMC_TIMEOUT_INFINITE to specify that the function should never return due to a timeout.

If the remote session does generate an event, the reasonCode indicates what specific event type occurred. The following table lists the valid values for the reasonCode return value:

| Value | Description |
|---|---|
| PXIMC_EVENT_ASSERTED | The remote session generated an event. |
| PXIMC_EVENT_INTERFACE_DOWN | The remote system with which the session was established can no longer be accessed. |
| PXIMC_EVENT_CONNECTION_CLOSED | The remote session closed its connection. |

The rules for how to interpret the meaning of receiving an PXIMC_EVENT_ASSERTED event should be defined by the protocolNumber that was used to initiate the connection.

### Implementation Requirements

**RULE**: The event queue on a session SHALL be implemented as a one-deep queue of events, with no notification of overflow of the event queue.

**OBSERVATION**: It is possible that PXIMC_EVENT_ASSERTED events are discarded if one of the paired sessions calls PXIMC_assertEvent multiple times while the other session does not call PXIMC_waitForSessionEvent.

**OBSERVATION**: If an event occurs while a process is not waiting for events, the next time the process waits for an event it will be notified that an event occurred.

**OBSERVATION**: When a process calls PXIMC_waitForSessionEvent it will be notified of the last event that occurred on the session since the last time the same process called PXIMC_waitForSessionEvent.

**OBSERVATION**: Callers of PXIMC_waitForSessionEvent should only call this function form one thread per process for a given session. Calling this function from multiple threads will result in only one of the threads being notified of any specific event.

**RULE**: If the paired session closes its connection, the PXIMC_EVENT_CONNECTION_CLOSED event SHALL be the next pending event to be returned through PXIMC_waitForSessionEvent.

**RULE**: IF the interface is removed from the system AND there is no pending PXIMC_EVENT_CONNECTION_CLOSED event, the application SHALL be notified by providing an PXIMC_EVENT_CONNECTION_CLOSED event to be the next pending event to be returned through PXIMC_waitForSessionEvent. In the case of interface removal, the PXIMC_EVENT_CONNECTION_CLOSED event is manually created by the PXImc vendor-specific layer, and was not generated by the remote session calling PXIMC_closeWindow.

**RULE**: IF an PXIMC_EVENT_CONNECTION_CLOSED event is pending, THEN it SHALL never be overwritten by any other event type.

**RULE**: reasonCode SHALL NOT be written unless PXIMC_waitForSessionEvent returns PXIMC_SUCCESS.

**OBSERVATION**: A value of zero can be passed to timeoutInMilliseconds to poll on whether a session event has already occurred.

## 3.3.5 Closing a Session

### 3.3.5.1 PXIMC_closeWindow

### Purpose

Closes a PXImc window.

### Parameters

| Name | Direction | Type | Description |
|---|---|---|---|
| sessionNumber | In | uint32_t | The session to close. |

### Return Values

| Completion Code | Description |
|---|---|
| PXIMC_SUCCESS | The session has been successfully paired with a remote session. |

| Error Codes | Description |
|---|---|
| PXIMC_INVALID_SESSION | The session specified by sessionNumber is not an open session. |

## Description

PXIMC_closeWindow closes the PXImc specified by sessionNumber. PXIMC_closeWindow should be used to close any open session, regardless of its current state. If called on a session that has been paired with a remote session, PXIMC_closeWindow cleans up the resources used by the connection, notifies the remote connection through an PXIMC_EVENT_CONNECTION_CLOSED event, and unpairs the two sessions. If called on a session that has not been paired, PXIMC_closeWindow revokes the request issued through the window request (this requested window will no longer be evaluated for pairing when remote sessions are created).

Calling PXIMC_closeWindow indicates to the PXImc software that this connection is no longer desired.

**RULE**: At the time PXIMC_closeWindow is called, the addresses returned by PXIMC_waitForConnection MUST no longer be referenced or used in any way.

**RULE**: At the time PXIMC_closeWindow is called, the addresses returned by PXIMC_getPhysicalAddress MUST no longer be referenced or used in any way.

If either of the above two rules are violated, either the remote or local session may become unstable. Results of violating the above two rules are undefined.

**OBSERVATION**: The above rules apply to all threads using the PXImc connection. It is advised that all threads or hardware processes performing transactions across the PXImc connection be idle prior to calling PXIMC_closeWindow.

**OBSERVATION**: If a physicalAddress value was provided during the window request, calling PXIMC_closeWindow does not guarantee the remote session will have stopped accessing the physical address. The remote session must call PXIMC_closeWindow before the physical address will no longer be accessed.

**OBSERVATION**: The local side of the session can not be cleaned up and its resources marked as available until the local side calls PXIMC_closeWindow, even if the local side receives an PXIMC_EVENT_CONNECTION_CLOSED event. If the local session receives an PXIMC_EVENT_CONNECTION_CLOSED, it should still call PXIMC_closeWindow.

## Implementation Requirements

**OBSERVATION**: One of the two sessions calling PXIMC_closeWindow doesn't completely free either the local window or the remote window. Implementations of PXImc MUST keep the resources allocated to the connection marked as in-use and unavailable until both the local and remote sessions have called PXIMC_closeWindow.

**OBSERVATION**: It is likely that some users of the PXImc API will not call PXIMC_closeWindow. PXIMC_closeWindow MAY be implicitly called on the user's behalf if the process unloads the shared library that implements the PXImc API, or if the process terminates.

**RULE**: The only cases where `PXIMC_closeWindow` MAY be automatically called are:
- If the process unloads the shared library that implements the PXImc API
- If the process terminates

**RULE**: `PXIMC_closeWindow` SHALL NOT wait for the remote session to close before returning.

**OBSERVATION**: A window request after a `PXIMC_closeWindow` may fail until the remote session has closed its session.

**OBSERVATION**: It is not possible to guarantee the user calls `PXIMC_closeWindow`. On process termination, the vendor-specific kernel component must guarantee that resources allocated to the terminated process are cleaned up.

## 3.3.5.2 PXIMC_cleanup

### Purpose

Cleans up the PXImc environment, and closes any remaining open sessions.

### Parameters

None.

### Return Values

| Completion Code | Description |
|---|---|
| `PXIMC_SUCCESS` | The PXImc environment was cleaned up. |

### Description

Allows the PXImc libraries to close all remaining open sessions that had been previously opened by this process, and also destroy any internal data structures that may have been allocated during execution.

This function should only be called once all sessions have stopped being used. Any use of any data provided by any function after calling this function is undefined in behavior.

**OBSERVATION**: `PXIMC_cleanup` does not block waiting for the remote system to close all of its open sessions.

**OBSERVATION**: PXImc activity can be re-established after calling `PXIMC_cleanup` by calling `PXIMC_findInterfaces`.

# 4. PXImc Shared Component: PXImc Dispatcher

## 4.1 Overview

This section describes the concept of implementing a PXImc Dispatcher, where a user application can call into the PXImc Dispatcher, and the PXImc Dispatcher would make calls to a specific vendor's implementation of the PXImc API as necessary. The existence of the PXImc Dispatcher allows applications that use the PXImc API to be written to only deal with loading and interfacing with the shared PXImc Dispatcher, and therefore these applications will be completely portable to allow any PXImc Logic Block vendor to provide the actual underlying interface.

The PXImc Dispatcher includes a header file (`pximc.h`), a dynamic library (`pximc32.dll/pximc64.dll` on Windows variants, `libpximc32.so/libpximc64.so` on Solaris/Linux), and an import library for compilation (`pximc32.lib/pximc64.lib` on Windows variants) if needed. The PXImc Dispatcher implements all API functions defined in Section 3.3, *API*, to call the corresponding vendor-specific function.

## 4.2 Objectives

The following are the objectives of the PXImc Dispatcher:

- Maximize the portability and interoperability of applications to multiple PXImc Logic Block Vendors' implementations beyond the benefits of implementing the standardized PXImc API.
- An application using a PXImc interface should be able to be developed in such a way that any PXImc Logic Block vendor could supply the interface and the application would continue to function without modification/recompilation.
- Multiple PXImc Logic Block vendors should be able to co-exist on any system, and the coexistence should be transparent to client applications
- An application using PXImc should be able to be developed in such a way that the application can be unaware of the number of PXImc Logic Block vendors on the system, and be able to easily interact with any or all PXImc Logic Blocks, regardless of vendor.

## 4.3 Behavior

The algorithm implemented in the PXImc Dispatcher is that when it is loaded it iterates through all vendor-specific implementations that have registered with the PXImc Dispatcher as described in Section 4.4. The PXImc Dispatcher dynamically loads each of the vendor-specific registered implementations. The general behavior of the PXImc Dispatcher is to route calls to one or more appropriate vendor-specific user layer implementations. After a session is opened in the PXImc Dispatcher, all API calls that act on a session use the session number to map the PXImc Dispatcher session number to specific instance of a vendor-specific user layer and a session with that vendor-specific user layer. The behaviors of specific functions are defined below.

### 4.3.1 PXIMC_findInterfaces

The PXImc Dispatcher serially calls all registered vendor-specific user layer implementations. It combines the results of all implementations into a single array of interfaces.

**OBSERVATION**: The PXImc Dispatcher needs to perform some pointer manipulation, and return the sum of interfaces in `actualNumberOfInterfaces`.

**RULE**: The PXImc Dispatcher SHALL maintain a mapping of interfaces to the vendor-specific implementations that provided the interface.

**RULE**: The PXImc Dispatcher SHALL return an error value if any of the vendor-specific implementations returned an error. The error value returned SHALL be the first error returned by any vendor-specific implementation.

**RULE**: IF no PXImc vendor-specific implementations are registered with the PXImc Dispatcher, THEN the PXImc Dispatcher SHALL return `PXIMC_NO_PROVIDER`.

**RULE**: The PXImc Dispatcher SHALL guarantee that all `interfaceID` values returned by `PXIMC_findInterfaces` are unique for a given process.

**OBSERVATION**: The PXImc Dispatcher MAY need to translate an `interfaceID` returned by a vendor-specific implementation into a new, unique `interfaceID` to return to the caller of `PXIMC_findInterfaces`.

**RULE**: If PXImc Dispatcher translates any `interfaceID` values, the PXImc Dispatcher SHALL maintain a mapping of `interfaceID` values returned from the PXImc Dispatcher, to vendor-specific `interfaceID` values, and the vendor-specific implementation that returned the value.

**RULE**: The PXImc Dispatcher SHALL NOT reuse interface numbers. If an interface is removed from the system, the interface number used by that interface SHALL NOT be used on any interface later added to the system.

### 4.3.2 Interface-based functions

The following RULES/RECOMMENDATIONS/OBSERVATIONS apply globally to all functions that have an interface value as an input parameter, but don't open a session. The functions that are included are: `PXIMC_queryInterfaceInformation`, `PXIMC_waitForInterfaceEvent`, `PXIMC_findWindows`, and `PXIMC_queryWindowInformation`.

The PXImc Dispatcher SHALL use the mapping generated in `PXIMC_findInterfaces` to directly call the vendor-specific implementation that supplied the interface.

**RULE**: If the `interfaceID` argument is not present in the mapping table, the PXImc Dispatcher SHALL internally refresh the mapping table using the same algorithm used to initially generate the table.

**RULE**: If after refreshing the mapping table, the `interfaceID` is still not present in the mapping table, the PXImc Dispatcher SHALL return `PXIMC_INVALID_INTERFACE`.

**RULE**: If the `interfaceID` is present in the mapping table, the PXImc Dispatcher SHALL return the value returned by vendor-specific implementation, and return all values of output parameters directly as returned by the vendor-specific implementation.

### 4.3.3 Requesting a window

The PXImc Dispatcher SHALL use the mapping generated in `PXIMC_findInterfaces` to directly call the vendor-specific implementation that supplied the interface.

**RULE**: If the `interfaceID` argument is not present in the mapping table, the PXImc Dispatcher SHALL internally refresh the mapping table using the same algorithm used to initially generate the table.

**RULE**: If after refreshing the mapping table, the `interfaceID` is still not present in the mapping table, the PXImc Dispatcher SHALL return `PXIMC_INVALID_INTERFACE`.

**RULE**: If the `interfaceID` is present in the mapping table, the PXImc Dispatcher SHALL return the value returned by vendor-specific implementation, and return all values of output parameters directly as returned by the vendor-specific implementation.

**RULE**: The PXImc Dispatcher SHALL maintain a mapping of `sessionNumber` to vendor-specific implementation that returned the `sessionNumber`.

**RULE**: If the vendor-specific implementation returned a value of `PXIMC_SUCCESS`, the PXImc Dispatcher SHALL add the returned `sessionNumber` to the mapping table.

**RULE**: The PXImc Dispatcher SHALL guarantee that all `sessionNumber` values returned by any window request are unique for a given process.

**OBSERVATION**: The PXImc Dispatcher MAY need to translate a `sessionNumber` returned by a vendor-specific implementation into a new, unique `sessionNumber` to return to the window requestor.

**RULE**: If PXImc Dispatcher translates any `sessionNumber` values, the PXImc Dispatcher SHALL maintain a mapping of `sessionNumber` values returned from the PXImc Dispatcher, to vendor-specific `sessionNumber` values, and the vendor-specific implementation that returned the value.

### 4.3.4 Session-based functions

The following RULES/RECOMMENDATIONS/OBSERVATIONS apply globally to all functions that have a `sessionNumber` value as an input parameter. The functions that are included are: `PXIMC_waitForConnection`, `PXIMC_getPhysicalAddress`, `PXIMC_waitForSessionEvent`, `PXIMC_assertEvent`, `PXIMC_closeWindow`.

**RULE**: If the `sessionNumber` value is not present in the mapping table generated by the window request, the PXImc Dispatcher SHALL return error `PXIMC_INVALID_SESSION`.

**RULE**: If the `sessionNumber` value is present in the mapping table generated by the window request, the PXImc Dispatcher SHALL replace the translated `sessionNumber` value with the vendor-specific `sessionNumber`, if the `sessionNumber` had been translated.

**RULE**: If the `sessionNumber` value is present in the mapping table generated by the window request, the PXImc Dispatcher SHALL directly call the corresponding function in the vendor-specific implementation that had returned the `sessionNumber`.

**RULE**: If the `sessionNumber` value is present in the mapping table generated by the window request, the PXImc Dispatcher SHALL return the value returned by the vendor-specific implementation, and return all values of output parameters directly as returned by the vendor-specific implementation.

When a session is closed by calling `PXIMC_closeWindow`, the `sessionNumber` closed can be removed from the mapping table.

### 4.3.5 PXIMC_cleanup

The PXImc Dispatcher serially calls all registered vendor-specific user layer implementations.

The PXImc Dispatcher can clear and free all of its internal mapping tables and data structures, including the `interfaceID` mapping table and the `sessionNumber` mapping table.

## 4.4 Registration

### 4.4.1 Windows

**RULE**: All vendor-specific PXImc implementations SHALL ensure that the PXImc Dispatcher is installed. If the PXImc Dispatcher is not installed, then the vendor-specific installer SHALL execute the PXImc Dispatcher installer.

**RULE**: Vendor-specific PXImc implementations SHALL NOT be named either `pximc32.dll` or `pximc64.dll` to avoid name collision with shared component.

#### 4.4.1.1 32-bit Windows

Vendor-specific PXImc implementations must create a new registry key to register as a PXImc compliant implementation.

The new registry key must be a subkey of

`HKEY_LOCAL_MACHINE\Software\PXISA\PXIMC\CurrentVersion\`

The name of the new subkey SHALL be the vendor-ID assigned to the PXImc Logic Block vendor by the PCI-SIG.

**RULE**: If the following registry key is not present:

`HKEY_LOCAL_MACHINE\Software\PXISA\PXIMC\CurrentVersion\`

The vendor-specific installer SHALL NOT create it, and SHALL error out.

**OBSERVATION**: The previous rule dictates that the PXImc Dispatcher installer is executed before any vendor-specific installer.

The vendor-specific key SHALL contain the following fields:

A `REG_SZ` named `Location` that contains the absolute path to the vendor-specific PXImc dynamic library. The `Location REG_SZ` must contain a path to a 32-bit vendor-specific PXImc dynamic library.

A `REG_DWORD` named `Version` that contains the PXImc version implemented by this PXImc dynamic library, as defined in the header file. For example, a dynamic library that implements the 1.0 version of this specification would set its `Version` to 0x00010000.

#### 4.4.1.2 64-bit Windows

Vendor-specific PXImc implementations must create two new registry keys to register as a PXImc compliant implementation.

The first new registry key must be a subkey of

`HKEY_LOCAL_MACHINE\Software\PXISA\PXIMC\CurrentVersion\`

The name of the new subkey SHALL be the vendor-ID assigned to the PXImc Logic Block vendor by the PCI-SIG.

The second new registry key must be a subkey of

`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PXISA\PXIMC\CurrentVersion\`

The name of the new subkey SHALL be the vendor-ID assigned to the PXImc Logic Block vendor by the PCI-SIG.

**RULE**: If the following registry keys are not present:

`HKEY_LOCAL_MACHINE\Software\PXISA\PXIMC\CurrentVersion\`

`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PXISA\PXIMC\CurrentVersion\`

The vendor-specific installer SHALL NOT create it, and SHALL error out.

**OBSERVATION**: The previous rule dictates that the PXImc Dispatcher installer is executed before any vendor-specific installer.

The vendor-specific keys SHALL contain the following fields:

A `REG_SZ` named `Location` that contains the absolute path to the vendor-specific PXImc dynamic library.

A `REG_DWORD` named `Version` that contains the PXImc version implemented by this PXImc dynamic library, as defined in the header file. For example, a dynamic library that implements the 1.0 version of this specification would set its `Version` to 0x00010000.

The `Location REG_SZ` in `HKEY_LOCAL_MACHINE\Software\PXISA\PXIMC\CurrentVersion\` must contain a path to a 64-bit vendor-specific PXImc dynamic library.

The `Location REG_SZ` in `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PXISA\PXIMC\CurrentVersion\` must contain a path to a 32-bit vendor-specific PXImc dynamic library.

A vendor-specific installer MAY leave the value of either of the `Location REG_SZ` uninitialized to indicate no dynamic library exists for the specified environment.

**RULE**: The registry keys registering a vendor-specific PXImc implementation SHALL be removed if the vendor-specific implementation is removed or uninstalled from the system.

### 4.4.2 Linux

**RULE**: All vendor-specific PXImc implementations SHALL ensure that the PXImc Dispatcher is installed. If the PXImc Dispatcher is not installed, then the vendor-specific installer SHALL execute the PXImc Dispatcher installer.

**RULE**: All vendor-specific PXImc implementations SHALL NOT be named either `libpximc32.so` or `libpximc64.so` to avoid name collision with shared component.

If no vendor-specific implementations are registered when `PXIMC_findInterfaces` is called, all installed vendor-specific libraries in the appropriate library directory are linked and registered.

## 4.5 Installation

### 4.5.1 32 bit Windows

#### TERMS

The following terms are used in this section:

`<PROGRAMFILES>` is the Windows Program Files directory. The default location of this directory is `C:\Program Files`, but can be changed by the user at OS install time.

`<PXIMCPATH>` is the target directory for the PXImc Dispatcher components.

**RULE**: The PXImc Dispatcher installer SHALL be named “PXImc Shared Components” and SHALL have its own entry in the Windows Add/Remove Program list.

The value of `<PXIMCPATH>`:

IF the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\PXISA\PXIMC\CurrentVersion` exists and contains the value `PXIMCPATH`, and this value designates a directory that exists, THEN the value of `<PXIMCPATH>` SHALL be the value of this key’s `PXIMCPATH` string value

OTHERWISE, the default value SHALL be `<PROGRAMFILES>\PXISA\PXIMC`. The PXImc Dispatcher installer SHALL allow the user to change the value of `<PXIMCPATH>`.

**RULE**: After determining the value of `<PXIMCPATH>`, the PXImc Dispatcher installer SHALL create a `REG_SZ` named `PXIMCPATH` in the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\PXISA\PXIMC\CurrentVersion`, IF the `REG_SZ` did not previously exist. Its value SHALL be the path of `<PXIMCPATH>`.

**RULE**: The PXImc Dispatcher installer SHALL create any of the following directories that don’t already exist:

`<PXIMCPATH>`

`<PXIMCPATH>\WinNT`

**RULE**: The PXImc Dispatcher 32-bit installer SHALL install the following files, unless newer versions of the files are already installed:

`<PXIMCPATH>\WinNT\pximc.h`

`<PXIMCPATH>\WinNT\pximc32.lib`

`<PXIMCPATH>\pximc32.dll`

**RULE**: The PXImc Dispatcher installer SHALL add `<PXIMCPATH>` to the Windows PATH.

**RULE**: The PXImc Dispatcher installer SHALL create the following registry keys and values under `HKEY_LOCAL_MACHINE\Software` if they didn’t previously exist:

- `PXISA`
- `PXISA\PXIMC`
- `PXISA\PXIMC\CurrentVersion`

Value: InstallerVersion—the version number of the PXImc Dispatcher installer. If this previously existed, the value must be updated.

Value: PXIMCPATH—<PXIMCPATH>

**RULE**: The 32-bit PXImc Dispatcher installer SHALL NOT install on any 64-bit Windows operating system.

RULE: The PXImc Dispatcher installer SHALL require that the user has Administrative privileges.

RULE: The PXImc Dispatcher installer SHALL provide command line options to:
- Run silently (/q).
- Set <PXIMCPATH> (provide the command line argument "PXIMCPATHDIR=<custom path>").
- Repair the installation (/f).

**RULE**: The PXImc Dispatcher uninstaller SHALL detect, after removing the files it installed, whether any remaining files or non-empty folders remain. IF no files remain, THEN the uninstaller SHALL remove all directories and registry keys.

## 4.5.2 64 bit Windows

**TERMS**

The following terms are used in this section:

<PROGRAMFILES> is the Windows Program Files directory. The default location of this directory is C:\Program Files, but can be changed by the user at OS install time.

<PXIMCPATH> is the target directory for the 32-bit PXImc Dispatcher components.

<PXIMCPATH64> is the target directory for the 64-bit PXImc Dispatcher components.

**RULE**: The PXImc Dispatcher installer for 64-bit Windows SHALL be named "PXImc Shared Components 64-bit" and SHALL have its own entry in the Windows Add/Remove Program list.

**RULE**: The PXImc Dispatcher 64-bit installer SHALL install both the 32-bit PXImc Dispatcher, and the 64-bit PXImc Dispatcher.

The value of <PXIMCPATH64>:

IF the registry key HKEY_LOCAL_MACHINE\SOFTWARE\PXISA\PXIMC\CurrentVersion exists and contains the value PXIMCPATH, and this value designates a directory that exists, THEN the value of <PXIMCPATH64> SHALL be the value of this key's PXIMCPATH string value.

OTHERWISE, the default value SHALL be <PROGRAMFILES>\PXISA\PXIMC. The PXImc Dispatcher installer SHALL allow the user to change the value of <PXIMCPATH64>.

**RULE**: After determining the value of <PXIMCPATH64>, the PXImc Dispatcher installer SHALL create a REG_SZ named PXIMCPATH in the registry key HKEY_LOCAL_MACHINE\SOFTWARE\PXISA\PXIMC\CurrentVersion, IF the REG_SZ did not previously exist. Its value SHALL be the path of <PXIMCPATH64>.

The value of <PXIMCPATH>:

IF the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PXISA\PXIMC\CurrentVersion exists and contains the value PXIMCPATH, and this value designates a directory that exists, THEN the value of <PXIMCPATH> SHALL be the value of this key's PXIMCPATH string value

OTHERWISE, the default value SHALL be <PROGRAMFILES>\PXISA\PXIMC. The PXImc Dispatcher installer SHALL allow the user to change the value of <PXIMCPATH>.

**RULE**: After determining the value of <PXIMCPATH>, the PXImc Dispatcher installer SHALL create a REG_SZ named PXIMCPATH in the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PXISA\PXIMC\CurrentVersion, IF the REG_SZ did not previously exist. Its value SHALL be the path of <PXIMCPATH>.

**RULE**: The PXImc Dispatcher 64-bit installer SHALL create any of the following directories that don't already exist:

<PXIMCPATH>

<PXIMCPATH>\WinNT

<PXIMCPATH64>

<PXIMCPATH64>\Win64

**RULE**: The PXImc Dispatcher 64-bit installer SHALL install the following files, unless newer versions of the files are already installed:

<PXIMCPATH>\WinNT\pximc.h

<PXIMCPATH>\WinNT\pximc32.lib

<PXIMCPATH>\pximc32.dll

<PXIMCPATH64>\Win64\pximc.h

<PXIMCPATH64>\Win64\pximc64.lib

<PXIMCPATH64>\pximc64.dll

**RULE**: The PXImc Dispatcher installer SHALL add <PXIMCPATH> and <PXIMCPATH64> to the Windows PATH. If the value of <PXIMCPATH> and <PXIMCPATH64> are the same, only one value will be added to the Windows PATH.

**RULE**: The PXImc Dispatcher 64-bit installer SHALL create the following registry keys and values under HKEY_LOCAL_MACHINE\Software if they didn't previously exist:

- PXISA
- PXISA\PXIMC
- PXISA\PXIMC\CurrentVersion

  Value: InstallerVersion—the version number of the PXImc Dispatcher installer. If this previously existed, the value must be updated.

  Value: PXIMCPATH—<PXIMCPATH64>

**RULE**: The PXImc Dispatcher 64-bit installer SHALL create the following registry keys and values under HKEY_LOCAL_MACHINE\Software\Wow6432Node if they didn't previously exist:

- PXISA
- PXISA\PXIMC
- PXISA\PXIMC\CurrentVersion

Value: InstallerVersion—the version number of the PXImc Dispatcher installer. If this previously existed, the value must be updated.

Value: PXIMCPATH—<PXIMCPATH>

**RULE**: The 64-bit PXImc Dispatcher installer SHALL NOT install on any 32-bit Windows operating system.

**RULE**: The PXImc Dispatcher installer SHALL require that the user has Administrative privileges.

**RULE**: The PXImc Dispatcher installer SHALL provide command line options to:

- Run silently (/q).
- Set <PXIMCPATH64> and <PXIMCPATH> (provide the command line argument(s) "PXIMCPATH64DIR=<custom path>" and/or "PXIMCPATHDIR=<custom path>").
- Repair the installation (/f).

**RULE**: The PXImc Dispatcher uninstaller SHALL detect, after removing the files it installed, whether any remaining files or non-empty folders remain. IF no files remain, THEN the uninstaller SHALL remove all directories and registry keys.

**RULE**: The PXImc Dispatcher uninstaller SHALL notify the user if cleanup isn't completed.

### 4.5.3 Linux

The Linux implementation of the PXImc Dispatcher may be installed either as source or as shared libraries compiled for the particular distribution and version of Linux. In both cases all components are installed in the /opt/pximc directory hierarchy.

**RULE**: The include file, `/opt/pximc/include/pximc.h`, MUST be installed.

**RULE**: For source installations, the files `/opt/pximc/src/Makefile and /opt/pximc/src/pximc.c` must be installed. For binary installations, they MAY be installed.

**RULE**: The manpage style documentation file, `/opt/pximc/share/man/man3/pximc.3.gz`, must be installed.

#### 4.5.3.1 32-bit Linux

**RULE**: For source installations, `/opt/pximc/src/Makefile` MUST compile and install the library file, `/opt/pximc/lib/libpximc32.so`.

**RULE**: Binary installations MUST install the library file, `/opt/pximc/lib/libpximc32.so`.

**RULE**: All vendor specific PXImc libraries MUST install symbolic links to their library files in `/opt/pximc/lib`.

#### 4.5.3.2 64-bit Linux

**RULE**: For source installations, `/opt/pximc/src/Makefile` MUST compile and install both the 32-bit version of the library file, `/opt/pximc/lib32/libpximc32.so` and the 64-bit version of the library file, `/opt/pximc/lib64/pximc64.so`.

**RULE**: Binary installations MUST install both the 32-bit version of the library file, `/opt/pximc/lib32/libpximc32.so` and the 64-bit version of the library file, `/opt/pximc/lib64/pximc64.so`.

**RULE**: The directory, `/opt/pximc/lib`, MUST be symbolically linked to `/opt/pximc/lib64`.

**RULE**: All vendor specific PXImc libraries MUST install symbolic links to their 32-bit library files in `/opt/pximc/lib32` and symbolic links to their 64-bit library files in `/opt/pximc/lib64`.

**PERMISSION**: A vendor MAY supply only a 32-bit or a 64-bit version of a PXImc library file.

# 5. Protocols

## 5.1 Overview

Within a memory window, it must be possible for two paired sessions to be able to have a common interpretation of the data being transferred. Additionally, paired sessions need to have a common understanding of when to use `PXIMC_assertEvent`, and how to interpret a received `PXIMC_EVENT_ASSERTED`. This specification does not define the format of the data within the memory windows, but instead allows for flexibility for different interpretations. This specification also does not define specifically when a session should send events or what it should do when one is received, but allows for any usage that both sessions determine is ideal. The "protocol" of the connection defines how data in both the local window and remote window should be interpreted, and specifics of how events should be used. By requesting a window with a specific protocol value, the requester is agreeing to read and write data to the windows and use events as defined by the protocol.

**OBSERVATION**: The PXImc API does not manipulate in any way any data passed across the PXImc interface to either the local or remote window.

**OBSERVATION**: Because the PXImc API does not perform any data manipulation, the protocol layer needs to account for differences in endianness across the interface. For example, if one of the systems in the connection is big endian and the other is little endian, the protocol may need to manipulate the data before it can be correctly interpreted when passed between the two systems.

As part of their definition of using events, protocols must only rely on a 1-deep queue being used to hold events. A potential way to deal with event queuing is for the protocol to require the receiver of an event to somehow acknowledge the event to the sender of the event before the sender sends an additional event.

Sessions will be paired only if the two sessions both pass the same protocol value to their window request.

**OBSERVATION**: By only pairing sessions of like protocols, paired sessions are guaranteed to have the same interpretation of data within the memory window, and interpret event usage identically.

It is envisioned that this specification may be extended in the future to define some "standard" protocols. At this time, there are no protocols that are standardized.

As no standardized protocols exist, the only way to communicate over PXImc is using a proprietary protocol, where the rules of the protocol are specific to the processes using the protocol. To avoid protocol collisions, the following rules should be used for implementing proprietary protocols:

**RULE**: The protocols in the range 0xF0000000–0xFFFFFFFF are reserved for proprietary protocols.

**RULE**: The protocols in the range 0xFXXXX000–0xFXXXXFFF are reserved for a specific vendor, where 'XXXX' is the vendor-ID assigned by the PCI-SIG.

**RULE**: Users of PXImc SHALL NOT use protocol values other than those within the range assigned for their use by the above rule.

**RULE**: Allocation of the protocol numbers 0x00000000–0xEFFFFFFF will defined by future PXImc protocol specifications.

# 6. Virtual Mesh

## 6.1 Overview

The PXImc hardware specification defines the only valid physical topology to be a tree topology, with all nodes connecting to a central device (Primary System Host). Virtual Mesh is the software feature that allows two leaf nodes in the tree to send and receive data directly to and from each other. In Figure 6-1, allowing PXImc Device1 to interact with PXImc Device2 is an example of virtual mesh.

Additionally, while a form of "indirect" virtual mesh could theoretically be instantiated by creating a 1-to-*N* mapped shared memory segment located on the Primary System Host that is mapped to be accessible by, for example, both Device1 and Device2 in Figure 6-1, there is no requirement in this specification or facility in the API that support creating or using such a mapping.

Memory mappings in this specification are always 1-to-1 and only between the Primary System Host and individual PXImc Devices. In other words, two or more individual PXImc Devices can ONLY communicate with each other through individual communication of each with the Primary System Host and the PSH being configured to coordinate its individual communications with Devices appropriately.

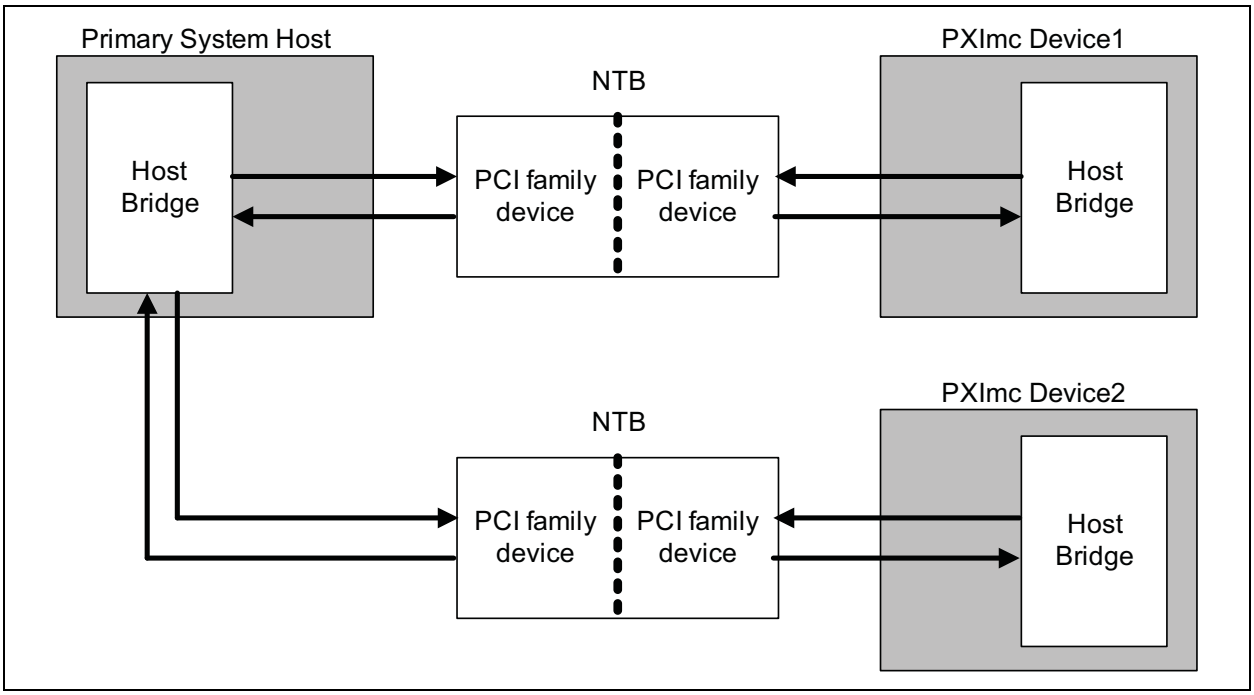


**Figure 6-1.** PXImc Tree Topology

Due to the technical complexity with supporting virtual mesh, support for this feature is not included in this specification. Adding virtual mesh to this revision of the specification would require a standardized way of publishing BAR addresses and sizes, interrupt register offsets and operation details, some resource manager to assign ownership of slices of BARs, and an algorithm for interacting with memory windows between the two devices. This complexity is significant, and is left for a future revision of the specification. A PXImc vendor may implement virtual mesh between PXImc Devices where it provides the PXImc hardware on all involved PXImc Devices. Such an implementation is beyond the scope of this version of the specification.

# A. Appendix: Example Use

This appendix is included to provide examples of how the PXImc API can be used to accomplish some typical use-cases. In each of the following use-cases, the following diagram will be used to help describe the scenario:

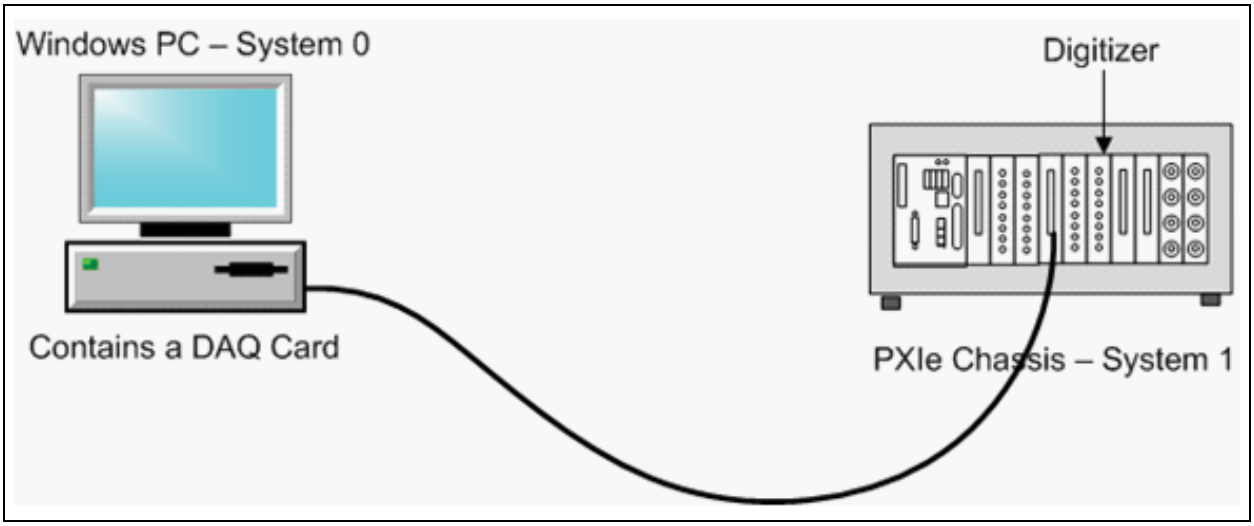


**Figure A-1.** Example Configuration

## A.1 Process to Process

Using the configuration shown in Figure A-1, a process on System 0 may want to communicate with a process on System 1 over PXImc. This section describes some common ways this connection can be established.

The PXImc connection between System 0 and System 1 is represented by a unique interface. The interface between the two systems is returned on each system to the process by calling `PXIMC_findInterfaces`.

The process on System 0 and the process on System 1 must both be able to interpret data and events passed between the two processes in a uniform way. This interpretation of the data is indicated by the protocol used to communicate between the two processes. To establish a connection, both processes must use the same `protocolNumber`.

The processes can initialize a connection by either using a client/server relationship, or a peer/peer relationship. If using the client/server relationship, one of the two processes must act as the “server”, being the first to initiate the connection. The other must then act as the “client”, initiating the connection after the server has initiated its portion of the connection. If using a peer/peer relationship, both sides initiate the

connection as a “peer”, and either process may initiate the connection prior to the other. The only differences between client/server and peer/peer initialization models are the behaviors of the request window calls—once the connection has been established both models are equivalent.

Each process must determine what its requirements are for local and remote windows. A process that only writes may need only a remote window (and therefore pass zero for the two local window size parameters). A process that reads and writes may need both a local and remote window (and therefore pass non-zero values for both window min size parameters). The appropriate size to request depends on the amount of data to be passed to the other process, and how quickly the paired session will consume the data.

Process-to-process connections must use Logical window connection type.

The `windowData` parameter is used to describe characteristics of the local process to the remote system. The data passed in the `windowData` is completely up to the process. One example of data that could be passed in the `windowData` is a string that describes the function of the process `windowData` values are only used when connecting as a “server” or as a “peer”. These are the only types of connections that can be successfully requested without a paired session being immediately available. The data provided in the `windowData` can be read on the remote system by calling `PXIMC_findWindows`, and then `PXIMC_queryWindowInformation`, querying attribute `PXIMC_U8_WINDOW_DATA`. The `windowData` is a mechanism for describing the role or functionality of the server or peer session to potential client/peer sessions prior to the client/peer initiating a connection.

The interface returned by `PXIMC_findInterfaces` can be used to request a memory window on that interface. The type of window request and parameters passed to it determine the pairing behaviors, as described above.

Both the System 1 and System 0 processes should call `PXIMC_waitForConnection` after requesting the window. If `PXIMC_waitForConnection` returns with status `PXIMC_SUCCESS`, then the session has been successfully paired. This means that the local request matched a remote request, and that the two sessions were paired so that they can communicate. `PXIMC_waitForConnection` returns addresses mapped to user-mode process addresses that the process can use to communicate with the remote process, using the rules to interpret the data that are dictated by the protocol used to initiate the connection.

For a specific example, assume a process on System 1 will act as the “server” of the connection, and a process on System 0 will act as the “client”. Both processes desire both a local and remote window of 4KB, but no smaller than 1KB. Both processes will use the same protocol. The server process will describe itself to the client process using the string “System 1 Server Process”.

In this example, the System 1 process calls `PXIMC_requestWindowLogicalAsServer`, and passes in the parameters described above. After calling `PXIMC_requestWindowLogicalAsServer`, the System 1 process calls `PXIMC_waitForConnection` to wait until the local session is paired with a remote session.

After System 1 calls `PXIMC_requestWindowLogicalAsServer`, when the System 0 process calls `PXIMC_findWindows` the window opened by System 1 will be represented in the returned `windowIDs`. The details of the System 1 server window are accessible by querying the various attributes of the window by supplying the desired `attributeID` to `PXIMC_queryWindowInformation`. System 0 can examine the available server connections, looking for any server that has a `windowData` of "System 1 Server Process". It does this by calling `PXIMC_findWindows`, and then querying the `PXIMC_WINDOW_CONNECTION_TYPE` and `PXIMC_U8_WINDOW_DATA` attributes of each of the windows. It then can use the values of the other attributes

to request its window. System 0 calls `PXIMC_requestWindowLogicalAsClient` using the values from `PXIMC_findWindows` and `PXIMC_queryWindowInformation`, and then calls `PXIMC_waitForConnection`.

The System 1 call to `PXIMC_waitForConnection` will return after the System 0 process requests its window. This indicates that the session pairing has completed with a paired session being located. The `PXIMC_waitForConnection` call on System 1 provides a `mappedRemoteAddress` and a `mappedLocalAddress` that the System 1 process can use to communicate with the paired System 0 process.

The System 0 processes call of `PXIMC_waitForConnection` will return immediately, because the session pairing was completed when System 0 requested its window. Like System 1, System 0 receives a `mappedLocalAddress` and `mappedRemoteAddress` to communicate with the System 1 process.

A sequence diagram depicting the connection initialization is provided in Figure A-2, *Process to Process Connection Initiation* (numbers in double brackets indicate detailed descriptions are provided in Table A-1.)

Both processes can use `PXIMC_assertEvent` and `PXIMC_waitForSessionEvent` to cause and service events between the two processes. A likely use of an event is to indicate that a data transmission is complete. A possible example of usage of `PXIMC_assertEvent` and `PXIMC_waitForSessionEvent` is provided in Figure A-3.

Once either process has completed its use of the connection, it can call `PXIMC_closeWindow` to terminate the connection on its side, and to notify the remote session that the connection has been closed. When either process receives an `PXIMC_EVENT_CONNECTION_CLOSED` event, it indicates the remote session closed the connection. At this time the local process should stop sending/receiving data and also call `PXIMC_closeWindow` to complete the process of terminating the connection. One possible way the connection could be terminated is shown in Figure A-4.

**Note** In the following figures, the numbers in double brackets indicate detailed descriptions that are provided in the table that follows each figure.

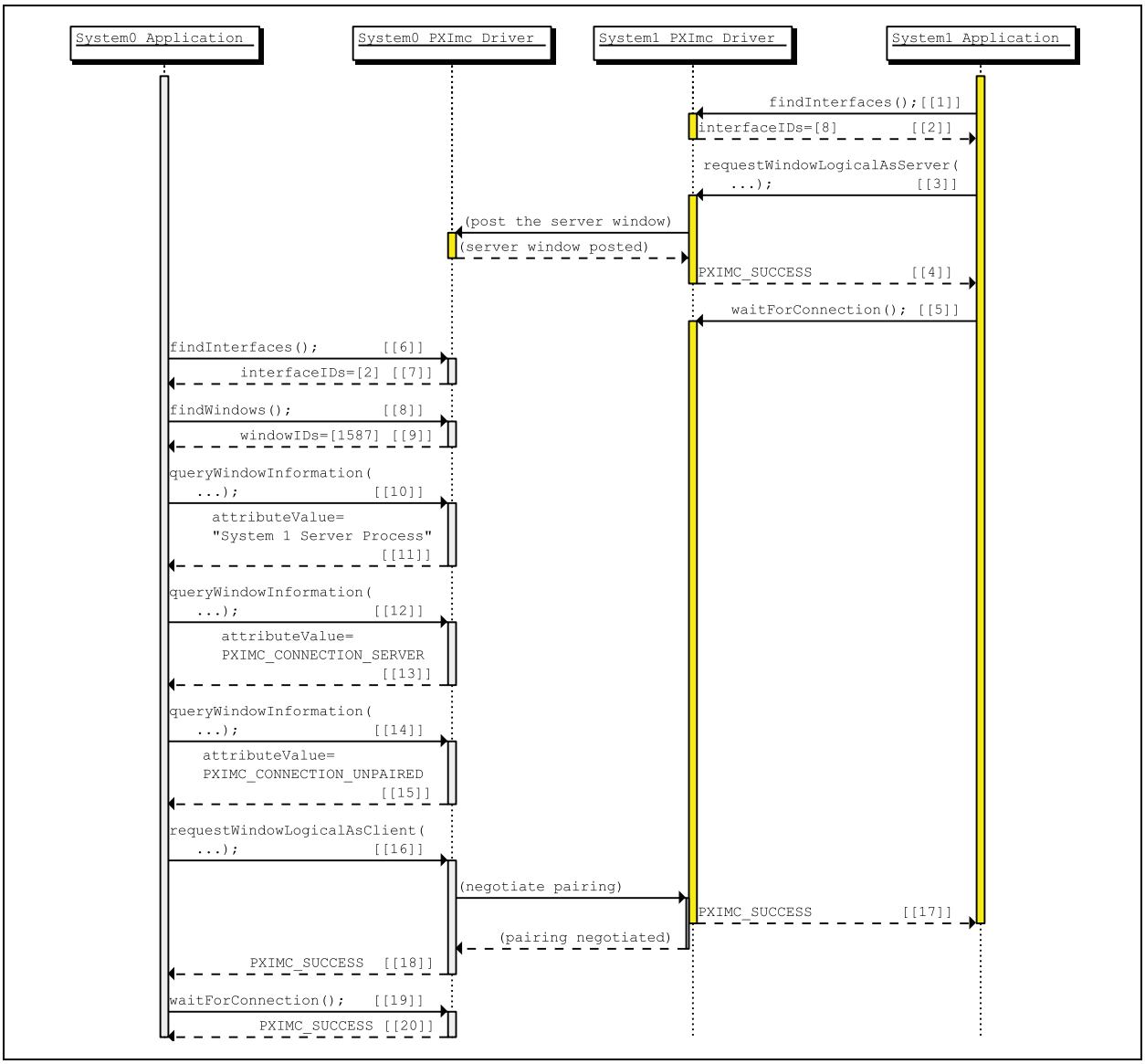


**Figure A-2.** Process to Process Connection Initiation

**Table A-1.**Figure A-2 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_findInterfaces (`<br>`  100, //maxNumberOfInterfaces`<br>`  interfaceIDsOut, //interfaceIDs`<br>`  numInterfacesOut`<br>`  //actualNumberOfInterfaces`<br>`  );` | The System 1 Application calls `findInterfaces()`. The application allocated an array of 100 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[2]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 8<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘8’. |
| [[3]] | `PXIMC_requestWindowLogicalAsServer (`<br>`  8, //interfaceID,`<br>`  0xABCD1000, //protocolNumber`<br>`  0x1000, // maxLocalSize`<br>`  0x400, // minLocalSize`<br>`  0x1000, // maxRemoteSize`<br>`  0x400, // minRemoteSize`<br>`  1587, // uniqueIdentifier`<br>`  “System 1 Server Process”, // windowData`<br>`  23, // windowDataSize`<br>`  sessionNumberOut //sessionNumber`<br>`  );` | System 1 requests its server window. It requests this window against interface ID ‘8’, as that is the interface found with `findInterfaces`. It passes the value ‘23’ for the `windowDataSize` because its `windowData` is 23 bytes long. |
| [[4]] | {`PXIMC_requestWindowLogicalAsServer` returns}<br>`sessionNumberOut` is set to 15<br>return value is `PXIMC_SUCCESS` | The window request is successful. The `sessionNumber` ‘15’ corresponds with this window request. The server window request has been posted to the remote system. |
| [[5]] | `PXIMC_waitForConnection (`<br>`  15, // sessionNumber`<br>`  0xFFFFFFFF, // timeoutInMilliseconds`<br>`  remoteAddressOut, // mappedRemoteAddress`<br>`  remoteSizeOut, // remoteSizeInBytes`<br>`  localAddressOut, // mappedLocalAddress`<br>`  localSizeOut // localSizeInBytes`<br>`  );` | System 1 waits until its window request is paired with a compatible window request on the remote system. |

**Table A-1.**Figure A-2 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[6]] | `PXIMC_findInterfaces (`<br>`  20, //maxNumberOfInterfaces`<br>`  interfaceIDsOut, //interfaceIDs`<br>`  numInterfacesOut`<br>`  //actualNumberOfInterfaces`<br>`  );` | The System 0 Application calls `findInterfaces()`. The application allocated an array of 20 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[7]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 2<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is '2'. |
| [[8]] | `PXIMC_findWindows (`<br>`  2, // interfaceID`<br>`  100, // maxNumberOfWindowIDs`<br>`  windowIDsOut, // windowIDs,`<br>`  numWindowsOut // actualNumberOfWindowIDs`<br>`  );` | System 0 is locating windows present on the remote side of interface '2'. The application allocated an array of 100 U32s in `windowIDsOut`, and passed the array size in `maxNumberOfWindowIDs`. |
| [[9]] | {`PXIMC_findWindows` returns}<br>`windowIDsOut` is set to 1587<br>`numWindowsOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One window was found. Its window ID is '1587'. |
| [[10]] | `PXIMC_queryWindowInformation (`<br>`  2, // interfaceID`<br>`  1587, // windowID`<br>`  PXIMC_U8_WINDOW_DATA, // attributeID`<br>`  1024, // maxSizeOfAttributeValue`<br>`  windowDataOut, // attributeValue`<br>`  sizeOut,`<br>`  //actualSizeOfAttributeValue`<br>`  );` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the window data of this window. The application allocated a 1024-byte buffer at `windowDataOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[11]] | {`PXIMC_queryWindowInformation` returns}<br>`windowDataOut` is set to "System 1 Server Process"<br>`sizeOut` is set to 23<br>return value is `PXIMC_SUCCESS` | The address `windowDataOut` now contains the `windowData`. The `sizeOut` value means that the `windowData` is 23 bytes long. |

**Table A-1.**Figure A-2 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[12]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`1587, // windowID`<br>`PXIMC_U32_WINDOW_CONNECTION_TYPE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`connectionTypeOut, // attributeValue`<br>`sizeOut, //actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the connection type of this window. The application allocated a single U32, 4 bytes, at `connectionTypeOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[13]] | {`PXIMC_queryWindowInformation` returns}<br>`connectionTypeOut` is set to `PXIMC_CONNECTION_SERVER`<br>`sizeOut` is set to 4<br>return value is `PXIMC_SUCCESS` | `connectionTypeOut` now contains the connection type value. |
| [[14]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`1587, // windowID`<br>`PXIMC_U8_WINDOW_PAIRING_STATE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`pairingStateOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the pairing state of this window. The application allocated a single U32, 4 bytes, at `pairingStateOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[15]] | {`PXIMC_queryWindowInformation` returns}<br>`pairingStateOut` is set to `PXIMC_WINDOW_UNPAIRED`<br>`sizeOut` is set to 4<br>return value is `PXIMC_SUCCESS` | `pairingStateOut` now contains the connection type value. |

**Table A-1.**Figure A-2 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[16]] | `PXIMC_requestWindowLogicalAsClient (`<br>`2, //interfaceID,`<br>`0xABCD1000, //protocolNumber`<br>`0x1000, // maxLocalSize`<br>`0x400, // minLocalSize`<br>`0x1000, // maxRemoteSize`<br>`0x400, // minRemoteSize`<br>`1587, // uniqueIdentifier`<br>`sessionNumberOut //sessionNumber`<br>`);` | After examining the attributes of window '1587', System 0 wants to attempt to connect to it. It requests a window against interface 2, window 1587. |
| [[17]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to process memory-mapped pointer of the remote window<br>`remoteSizeOut` is set to set to the actual size of the remote window<br>`localAddressOut` is set to process memory-mapped pointer of the local window<br>`localSizeOut` is set to the actual size of the local window<br>return value is `PXIMC_SUCCESS` | System 1's connection has been established once System 0 requested a compatible window, and the PXImc interface paired the System 0 and System 1 window requests. System 1 gets two pointers back, `remoteAddressOut` and `localAddressOut`, that it can use to access the local and remote window. |
| [[18]] | {`PXIMC_requestWindowLogicalAsClient` returns}<br>`sessionNumberOut` is set to 4<br>return value is `PXIMC_SUCCESS` | The window request on System 0 is successful. The `sessionNumber` '4' corresponds with this window request. The client window request has been paired with a compatible server. |

**Table A-1.**Figure A-2 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[19]] | `PXIMC_waitForConnection (`<br>`4, // sessionNumber`<br>`0x0, //timeoutInMilliseconds`<br>`remoteAddressOut,// mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 0 waits until its window request is paired with a compatible window request on the remote system. Because its client window request was successful, it knows it has already been paired with a compatible server, but needs to call `waitForConnection` to get the pointers to the local and remote windows. |
| [[20]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to process memory-mapped pointer of the remote window<br>`remoteSizeOut` is set to set to the actual size of the remote window<br>`localAddressOut` is set to process memory-mapped pointer of the local window<br>`localSizeOut` is set to the actual size of the local window<br>return value is `PXIMC_SUCCESS` | System 0's `waitForConnection` returns. System 0 gets two pointers back, `remoteAddressOut` and `localAddressOut`, that it can use to access the local and remote window. |

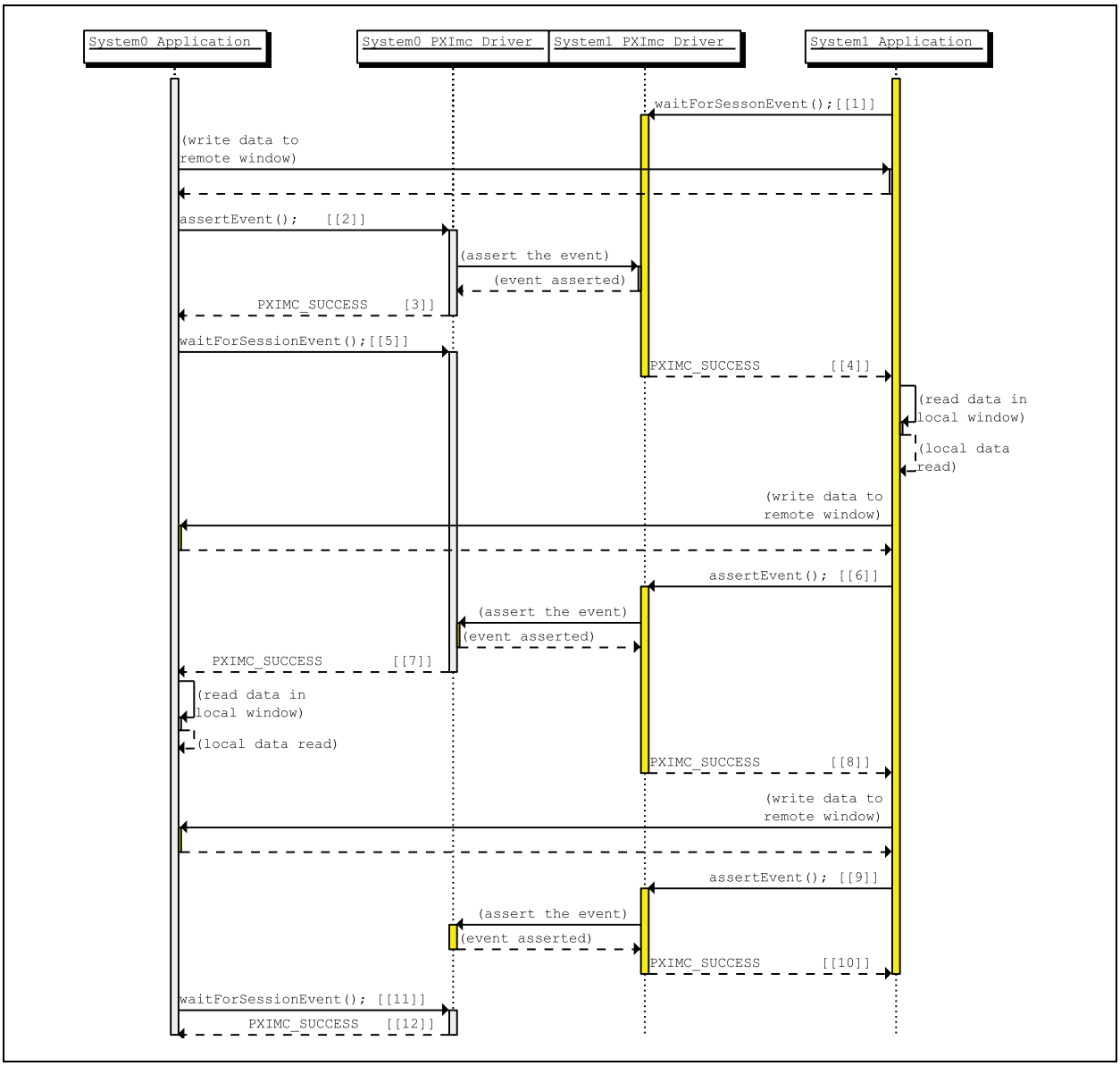


**Figure A-3.**Sample I/O and Events

**Table A-2.**Figure A-3 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_waitForSessionEvent (`<br>`15, //sessionNumber`<br>`PXIMC_TIMEOUT_INFINITE, //timeoutInMilliseconds`<br>`reasonCodeOut //reasonCode`<br>`);` | The System 1 Application calls `waitForSessionEvent()`.The application wants to wait until the System 0 Application calls `assertEvent()`. Because `PXIMC_TIMEOUT_INFINITE` was specified, `waitForSessionEvent` will not return until some event occurs. |
| [[2]] | `PXIMC_assertEvent (`<br>`4, //sessionNumber`<br>`);` | System 0 calls `assertEvent()`, potentially indicating that System 0 has completed writing some data, and it is now available for the System 1 Application to read. |
| [[3]] | {`PXIMC_assertEvent` returns}<br>return value is `PXIMC_SUCCESS` | `assertEvent()` returns successfully. |
| [[4]] | {`PXIMC_waitForSessionEvent` returns}<br>`reasonCodeOut` is set to `PXIMC_EVENT_ASSERTED`<br>return value is `PXIMC_SUCCESS` | An event was received by System 1, allowing `waitForSessionEvent()` to return. |
| [[5]] | `PXIMC_waitForSessionEvent (`<br>`4, //sessionNumber`<br>`PXIMC_TIMEOUT_INFINITE, //timeoutInMilliseconds`<br>`reasonCodeOut //reasonCode`<br>`);` | The System 0 Application calls `waitForSessionEvent()`.The application wants to wait until the System 1 Application calls `assertEvent()`. Because `PXIMC_TIMEOUT_INFINITE` was specified, `waitForSessionEvent` will not return until some event occurs. |
| [[6]] | `PXIMC_assertEvent (`<br>`15, //sessionNumber`<br>`);` | System 1 calls `assertEvent()`, potentially indicating that System 1 has completed writing some data, and it is now available for the System 0 Application to read. |
| [[7]] | {`PXIMC_waitForSessionEvent` returns}<br>`reasonCodeOut` is set to `PXIMC_EVENT_ASSERTED`<br>return value is `PXIMC_SUCCESS` | An event was received by System 0, allowing `waitForSessionEvent()` to return. |
| [[8]] | {`PXIMC_assertEvent` returns}<br>return value is `PXIMC_SUCCESS` | `assertEvent()` returns successfully. |

**Table A-2.**Figure A-3 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[9]] | `PXIMC_assertEvent (`<br>`15, //sessionNumber`<br>`);` | System 1 calls `assertEvent()`, potentially indicating that System 1 has completed writing some data, and it is now available for the System 0 Application to read. |
| [[10]] | {`PXIMC_assertEvent` returns}<br>return value is `PXIMC_SUCCESS` | `assertEvent()` returns successfully. Note that it isn't required for System 0 to have received the event before `assertEvent` can return. |
| [[11]] | `PXIMC_waitForSessionEvent (`<br>`4, //sessionNumber`<br>`PXIMC_TIMEOUT_INFINITE, //timeoutInMilliseconds`<br>`reasonCodeOut //reasonCode`<br>`);` | The System 0 Application calls `waitForSessionEvent()`. The application wants to wait until the System 1 Application calls `assertEvent()`. Because `PXIMC_TIMEOUT_INFINITE` was specified, `waitForSessionEvent` will not return until some event occurs. |
| [[12]] | {`PXIMC_waitForSessionEvent` returns}<br>`reasonCodeOut` is set to `PXIMC_EVENT_ASSERTED`<br>return value is `PXIMC_SUCCESS` | An event was received by System 0, allowing `waitForSessionEvent()` to return. The event had been received prior to System 0 calling `waitForSessionEvent()`, so it returned immediately. |

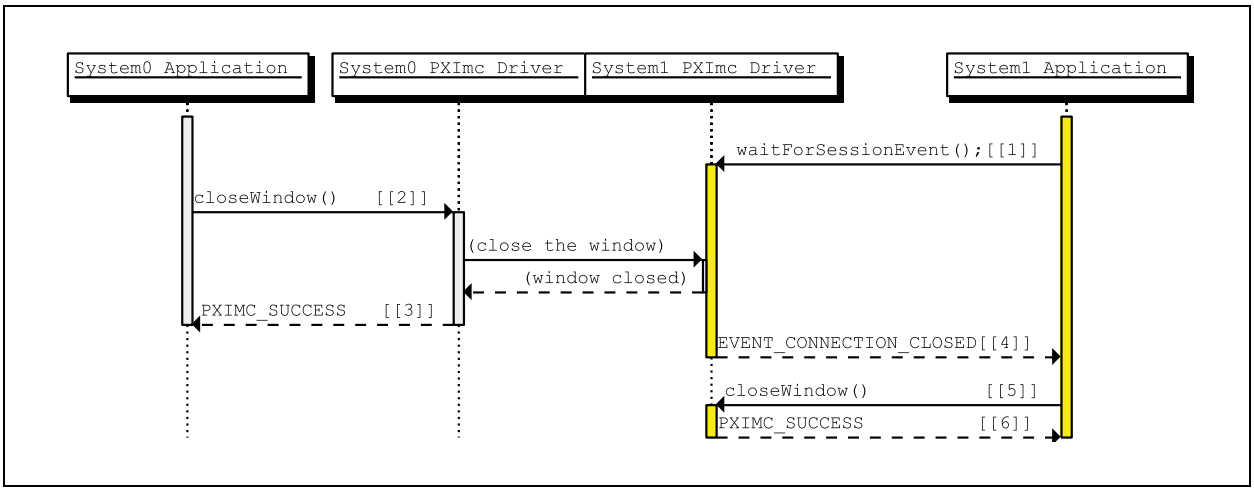


**Figure A-4.**Connection Termination

**Table A-3.**Figure A-4 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_waitForSessionEvent (`<br>`15, //sessionNumber`<br>`PXIMC_TIMEOUT_INFINITE, //timeoutInMilliseconds`<br>`reasonCodeOut //reasonCode`<br>`);` | The System 1 Application calls `waitForSessionEvent()`. The application wants to wait until the System 0 Application calls `assertEvent()`, or until another event occurs. Because `PXIMC_TIMEOUT_INFINITE` was specified, `waitForSessionEvent` will not return until some event occurs. |
| [[2]] | `PXIMC_closeWindow (`<br>`4, //sessionNumber`<br>`);` | System 0 Application closes it's window. It no longer wants to communicate with the System 1 Application. |
| [[3]] | {`PXIMC_closeWindow` returns}<br>return value is `PXIMC_SUCCESS` | System 0 Application's call to close the window returns successfully. |
| [[4]] | {`PXIMC_waitForSessionEvent` returns}<br>`reasonCodeOut` is set to `PXIMC_EVENT_CONNECTION_CLOSED`<br>return value is `PXIMC_SUCCESS` | An event was received by System 1, allowing `waitForSessionEvent()` to return. System 1 can no longer send any data to System 0. |
| [[5]] | `PXIMC_closeWindow (`<br>`15, //sessionNumber`<br>`);` | System 1 Application closes it's window. It must close its portion of the window to free resources on the interface. |
| [[6]] | {`PXIMC_closeWindow` returns}<br>return value is `PXIMC_SUCCESS` | System 1 Application's call to close the window returns successfully. |

## A.2 Process sourcing data directly to hardware

Using the configuration shown in Figure A-1, a process on System 0 may want to write data directly to the digitizer present in System 1 over PXImc. A common scenario where this type of connection is desired is if a process is sourcing data directly to an output device. This section describes a common way this connection can be established.

While there is only a single process directly participating in data transactions (the process on System 0), a process is also necessary on System 1 to establish the connection. After the connection is established, the System 1 process is idle. The PXImc connection between System 0 and System 1 is represented by a unique interface. The interface between the two systems is returned on each system to the process by calling `PXIMC_findInterfaces`.

The process on System 0 must have the same interpretation of the data as the digitizer in System 1. As the digitizer's interpretation of data is fixed by its register map, the System 0 process must share this data interpretation. The interpretation of the data is indicated by the protocol used to initiate the connection. To establish a connection, both processes must use the same protocol.

The processes can initialize a connection by using a client/server relationship. The System 1 process must act as the "server", being the first to initiate the connection. The System 0 process must then act as the "client", initiating the connection after the server has initiated its portion of the connection. The PXImc implementation will select the hardware resources that best meet the requirements of the connection. The physical resources used are determined based on the parameters of the connection request.

In the client/server initialization model, the System 1 process must be the first to request its window. The type of window in this example is a physical window location type, as the local target of the connection is a physical address that is manually supplied. The `protocolNumber` passed to the window request should be a protocol specific to the target hardware. For example, a protocol could be defined that specifies the specific digitizer's register map. This `protocolNumber` should be passed to the window request. The `localSize` should be set to the size of the digitizer's base address register that System 0 will be interacting with. The `physicalAddress` value should be set to the local physical address that will be targeted by System 0, which is the digitizer base address register in this case. The `uniqueIdentifier` can be set to zero to allow PXImc to assign any `uniqueIdentifier` to the request. The `windowData` can be set to any string that will enable the client software running on System 0 to specifically identify this window. For this example, System 1 will use "Vendor XYZ Digitizer 123".

The System 1 process is now ready to call `PXIMC_requestWindowPhysicalAsServer`. The interface returned by `PXIMC_findInterfaces` can be used to request a memory window on that interface. This request is made by calling `PXIMC_requestWindowPhysicalAsServer`. The parameters passed to `PXIMC_requestWindowPhysicalAsServer` determine the exact behaviors of the window request, as described above. The System 1 process calls `PXIMC_requestWindowPhysicalAsServer` using the values chosen above.

The process on System 1 then takes the returned `sessionNumber` and calls `PXIMC_waitForConnection` with a sufficiently large `timeoutInMilliseconds` value to block until the connection is complete. At this point, the connection is just requested, but a connection has not yet been established.

The process on System 0 can call `PXIMC_findWindows` to see all available windows running on System 1. After System 1 has successfully called `PXIMC_requestWindowPhysicalAsServer` as described above, the server window registered by Session 1 will be represented as one of the `windowIDs` returned from `PXIMC_findWindows`. The details of the System 1 server window are accessible by querying the various attributes of the window by supplying the desired `attributeID` to `PXIMC_queryWindowInformation`.

In this example, System 0 is looking for any `windowData` with the value "Vendor XYZ Digitizer 123" and `protocolNumber` that was defined for this digitizer's register map. System 0 can examine the available server connections, looking for any server with a specific `windowData` and a specific protocol number by calling `PXIMC_findWindows`, and then querying the `PXIMC_U8_WINDOW_DATA`, `PXIMC_WINDOW_CONNECTION_TYPE`, `PXIMC_WINDOW_PAIRING_STATE`, `PXIMC_WINDOW_LOCATION_TYPE`, and `PXIMC_WINDOW_PROTOCOL_NUMBER` attributes of each of the

windows. When it locates the window ID that has the attribute values it's looking for, it requests its window using the `windowID` for the `uniqueIdentifier` argument value. The System 0 request will use `PXIMC_requestWindowPhysicalAsClient`. For the other parameters, it passes the `protocolNumber` retrieved from `PXIMC_queryWindowInformation`, the exact window size retrieved from `PXIMC_queryWindowInformation` for both the `minRemoteSize` and `maxRemoteSize`.

The process on System 0 then takes the returned `sessionNumber` and calls `PXIMC_waitForConnection` with zero for the `timeoutInMilliseconds` parameter. As the System 0 session was paired with the System 1 session during the `PXIMC_requestWindowPhysicalAsClient` call, `PXIMC_waitForConnection` will successfully return, and it will supply the `mappedRemoteAddress` and `remoteSizeInBytes`.

The process on System 1 is notified that the connection has been established because its call to `PXIMC_waitForConnection` will return.

A sequence diagram depicting the connection initialization is provided in Figure A-6.

The only other role the System 1 process has is to clean up the connection when the connection is terminated. The System 1 process can call `PXIMC_waitForSessionEvent` with a long timeout, and when it receives an `PXIMC_EVENT_CONNECTION_CLOSED`, it should call `PXIMC_closeWindow`.

The System 0 process can now directly read from or write to the digitizer in System 1. If it writes data to the `mappedRemoteAddress`, the data written will be written to the System 1 `physicalAddress`.

After the System 0 process has completed interacting with the System 1 digitizer, it must call `PXIMC_closeWindow`. This will initiate clean up of the connection, and after the System 1 process also calls `PXIMC_closeWindow`, the connection will be terminated and its resources freed. An example of a way the connection could be terminated is shown in Figure A-4.

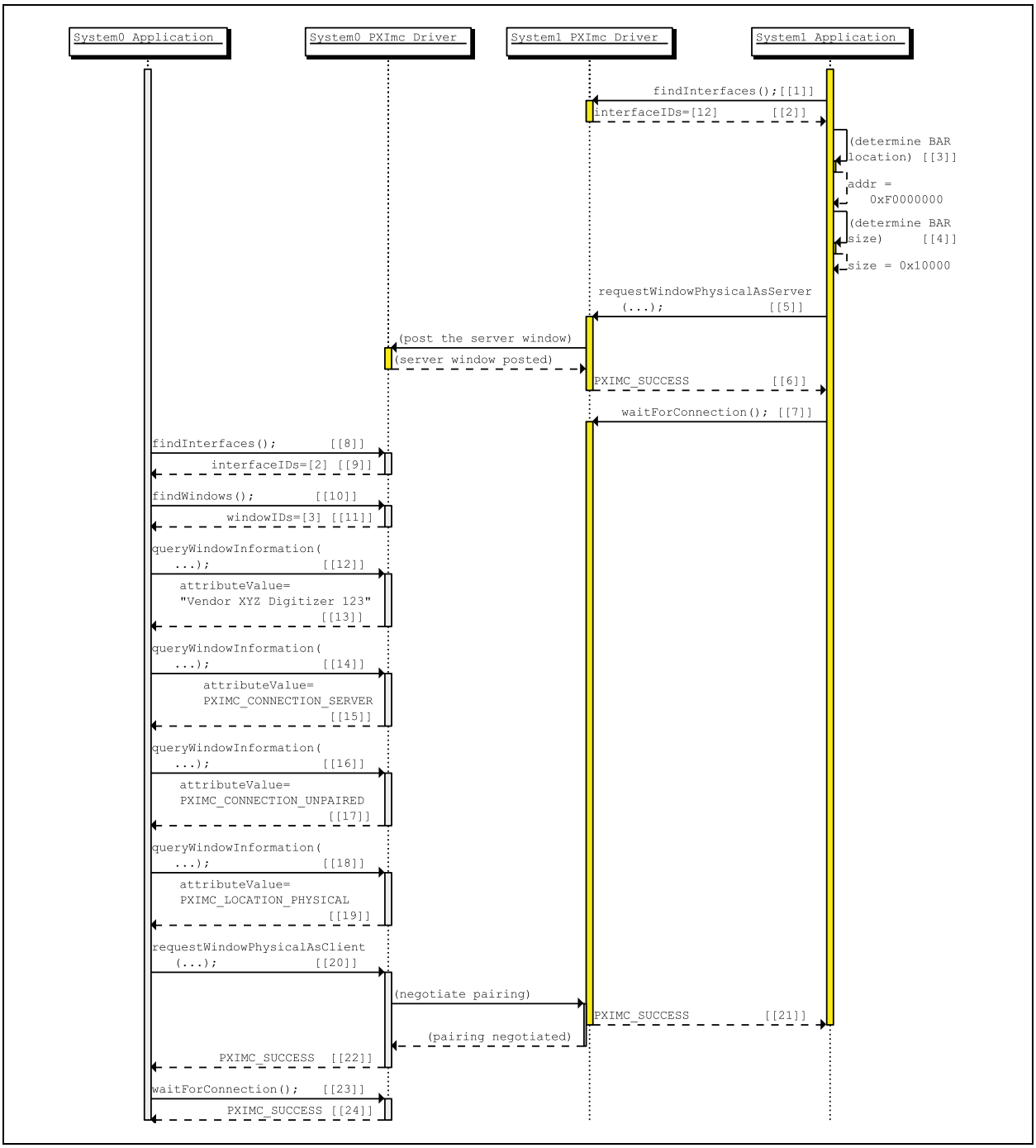


**Figure A-5.**Process to Hardware Connection Initiation

**Table A-4.**Figure A-5 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_findInterfaces (`<br>`  100, //maxNumberOfInterfaces`<br>`  interfaceIDsOut, //interfaceIDs`<br>`  numInterfacesOut`<br>`  //actualNumberOfInterfaces`<br>`  );` | The System 1 Application calls `findInterfaces()`. The application allocated an array of 100 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[2]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 12<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘12’. |
| [[3]] | {System 1 Application determines BAR address of the digitizer}<br>BAR Address = 0x00000000F0000000 | The means for the System 1 Application to determine the address of the digitizer BAR is beyond the scope of this specification. |
| [[4]] | {System 1 Application determines BAR size of the digitizer}<br>BAR Size = 0x10000 | The means for the System 1 Application to determine the size of the digitizer BAR is beyond the scope of this specification. |
| [[5]] | `PXIMC_requestWindowPhysicalAsServer (`<br>`  12, //interfaceID,`<br>`  0xABCD1000, //protocolNumber`<br>`  0x10000, // localSize`<br>`  0, // uniqueIdentifier`<br>`  0x00000000F0000000, //physicalAddress`<br>`  “Vendor XYZ Digitizer 123”,`<br>`  // windowData`<br>`  24, // windowDataSize`<br>`  sessionNumberOut //sessionNumber`<br>`  );` | System 1 requests its server window. It requests this window against interface ID ‘12’, as that is the interface found with `findInterfaces`. It passes the value ‘24’ for the `windowDataSize` because its `windowData` is 24 bytes long. |
| [[6]] | {`PXIMC_requestWindowLogicalAsServer` returns}<br>`sessionNumberOut` is set to 15<br>return value is `PXIMC_SUCCESS` | The window request is successful. The `sessionNumber` ‘15’ corresponds with this window request. The server window request has been posted to the remote system. |

**Table A-4.**Figure A-5 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[7]] | `PXIMC_waitForConnection (`<br>`15, // sessionNumber`<br>`0xFFFFFFFF, // timeoutInMilliseconds`<br>`remoteAddressOut, // mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 1 waits until its window request is paired with a compatible window request on the remote system. |
| [[8]] | `PXIMC_findInterfaces (`<br>`20, //maxNumberOfInterfaces`<br>`interfaceIDsOut, //interfaceIDs`<br>`numInterfacesOut`<br>`//actualNumberOfInterfaces`<br>`);` | The System 0 Application calls `findInterfaces()`. The application allocated an array of 20 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[9]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 2<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘2’. |
| [[10]] | `PXIMC_findWindows (`<br>`2, // interfaceID`<br>`100, // maxNumberOfWindowIDs`<br>`windowIDsOut, // windowIDs,`<br>`numWindowsOut // actualNumberOfWindowIDs`<br>`);` | System 0 is locating windows present on the remote side of interface ‘2’. The application allocated an array of 100 U32s in `windowIDsOut`, and passed the array size in `maxNumberOfWindowIDs`. |
| [[11]] | {`PXIMC_findWindows` returns}<br>`windowIDsOut` is set to 3<br>`numWindowsOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One window was found. Its window ID is ‘3’. |

**Table A-4.**Figure A-5 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[12]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U8_WINDOW_DATA, // attributeID`<br>`1024, // maxSizeOfAttributeValue`<br>`windowDataOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '3' on interface '2'. It is getting the window data of this window. The application allocated a 1024-byte buffer at `windowDataOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[13]] | {`PXIMC_queryWindowInformation` returns}<br>`windowDataOut` is set to "Vendor XYZ Digitizer 123"<br>`sizeOut` is set to 24<br>return value is `PXIMC_SUCCESS` | The address `windowDataOut` now contains the `windowData`. The `sizeOut` value means that the `windowData` is 24 bytes long. |
| [[14]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U32_WINDOW_CONNECTION_TYPE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`connectionTypeOut, // attributeValue`<br>`sizeOut, //actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '3' on interface '2'. It is getting the connection type of this window. The application allocated a single U32, 4 bytes, at `connectionTypeOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[15]] | {`PXIMC_queryWindowInformation` returns}<br>`connectionTypeOut` is set to `PXIMC_CONNECTION_SERVER`<br>`sizeOut` is set to 4<br>return value is `PXIMC_SUCCESS` | `connectionTypeOut` now contains the connection type value. |
| [[16]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U8_WINDOW_PAIRING_STATE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`pairingStateOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '3' on interface '2'. It is getting the pairing state of this window. The application allocated a single U32, 4 bytes, at `pairingStateOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |

**Table A-4.**Figure A-5 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[17]] | {PXIMC_queryWindowInformation returns}<br>pairingStateOut is set to PXIMC_WINDOW_UNPAIRED<br>sizeOut is set to 4<br>return value is PXIMC_SUCCESS | pairingStateOut now contains the connection type value. |
| [[18]] | PXIMC_queryWindowInformation (<br>2, // interfaceID<br>3, // windowID<br>PXIMC_U8_WINDOW_LOCATION_TYPE,<br>// attributeID<br>4, // maxSizeOfAttributeValue<br>pairingStateOut, // attributeValue<br>sizeOut,<br>//actualSizeOfAttributeValue<br>); | System 0 is getting information about windowID '3' on interface '2'. It is getting the location of this window. The application allocated a single U32, 4 bytes, at pairingStateOut, and is providing the size of the buffer in maxSizeOfAttributeValue. |
| [[19]] | {PXIMC_queryWindowInformation returns}<br>pairingStateOut is set to PXIMC_LOCATION_PHYSICAL<br>sizeOut is set to 4<br>return value is PXIMC_SUCCESS | pairingStateOut now contains the connection type value. |
| [[20]] | PXIMC_requestWindowPhysicalAsClient (<br>2, //interfaceID,<br>0xABCD1000, //protocolNumber<br>0x10000, // maxRemoteSize<br>0, // minRemoteSize<br>3, // uniqueIdentifier<br>sessionNumberOut //sessionNumber<br>); | After examining the attributes of window '3', System 0 wants to attempt to connect to it. It requests a window against interface 2, window 3. |
| [[21]] | {PXIMC_waitForConnection returns}<br>remoteAddressOut is set to NULL, as there is no remote window<br>remoteSizeOut is set to set to zero, as there is no remote window<br>localAddressOut is set to NULL, as the process cannot access the local window<br>localSizeOut is set to zero, as the process cannot access the local window<br>return value is PXIMC_SUCCESS | System 1's connection has been established once System 0 requested a compatible window, and the PXImc interface paired the System 0 and System 1 window requests. System 1 cannot use its window to perform any I/O, but it has enabled System 0 to access the digitizer BAR. |

**Table A-4.**Figure A-5 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[22]] | {`PXIMC_requestWindowLogicalAsClient` returns}<br>`sessionNumberOut` is set to 4<br>return value is `PXIMC_SUCCESS` | The window request on System 0 is successful. The `sessionNumber` '4' corresponds with this window request. The client window request has been paired with a compatible server. |
| [[23]] | `PXIMC_waitForConnection (`<br>`4, // sessionNumber`<br>`0x0, //timeoutInMilliseconds`<br>`remoteAddressOut,// mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 0 waits until its window request is paired with a compatible window request on the remote system. Because its client window request was successful, it knows it has already been paired with a compatible server, but needs to call `waitForConnection` to get the pointers to the remote window. |
| [[24]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to process memory-mapped pointer of the remote window (the digitizer BAR)<br>`remoteSizeOut` is set to set to the actual size of the remote window (the exact size of the digitizer BAR)<br>`localAddressOut` is set to NULL, as there is no local window<br>`localSizeOut` is set to zero, as there is no local window<br>return value is `PXIMC_SUCCESS` | System 0's `waitForConnection` returns. System 0 gets two pointers back, `remoteAddressOut` and `localAddressOut`, that it can use to access the local and remote window. `localAddressOut` will be NULL and `localSizeOut` will be zero, as there is no local window. When System 0 accesses `remoteAddressOut`, the result will be that the Digitizer BAR in System 1 is accessed. |

## A.3 Hardware sourcing data directly to process

Using the configuration shown in Figure A-1, a process on System 0 may want to receive data directly from the digitizer present in System 1 over PXImc. A common scenario where this type of connection is desired is if a process is syncing data directly from an input device. This section describes a common way this connection can be established.

While there is only a single process directly participating in data transactions (the process on System 0), a process is necessary on System 1 to establish the connection. After the connection is established, the System 1 process is idle. The PXImc connection between System 0 and System 1 is represented by a unique interface. The interface between the two systems is returned on each system to the process by calling `PXIMC_findInterfaces`.

The process on System 0 and the hardware on System 1 must both be able to interpret data passed between the two systems in a uniform way. This interpretation of the data is indicated by the protocol used to communicate between the hardware and the process. To establish a connection, both processes must use the same protocol.

The processes can initialize a connection by either using a client/server relationship, or a peer/peer relationship. If using the client/server relationship, one of the two processes must act as the "server", being the first to initiate the connection. The other must then act as the "client", initiating the connection after the server has initiated its portion of the connection. If using a peer/peer relationship, both sides initiate the connection as a "peer", and either process may initiate the connection prior to the other. The only difference between client/server and peer/peer initialization models is the behaviors of the window request itself—once the connection has been established both models are equivalent. The remainder of this example assumes the client/server initialization model is used. System 1 will initiate the connection as the "server", and System 0 as the "client".

The System 1 process must determine what its requirements are for local and remote windows. In this case, hardware on System 1 needs to write data to System 0. Therefore the System 1 process will request a remote window where `minRemoteSize` is the minimum remote window size that the hardware is able to use, and the `maxRemoteSize` is the ideal remote window size for the hardware source. It is possible that the `minRemoteSize` is equal to `maxRemoteSize`, meaning there's a specific size requirement for the remote window. In this example, System 1 does not need a local window, so it would set `minLocalSize` and `maxLocalSize` to zero.

This connection must use a logical window location type. This connection is a logical connection because the connection doesn't specify a specific physical address for the local window.

The `windowData` parameter is used to describe characteristics of the local process to the remote system. The data passed in the `windowData` is completely up to the process. One example of data that could be passed in the `windowData` is a string that describes the function of the process. `windowData` values should only be set by processes connecting as a "server" or as a "peer". These are the only types of connections that can be successfully requested without a paired session being immediately available. The data provided in the `windowData` can be read on the remote system by calling `PXIMC_queryWindowInformation`. In this example, System 1 might pass "ABC Vendor Digitizer source" for the `windowData`. The System 1 process may choose to set the `uniqueIdentifier` to zero, and allow the PXImc interface to select a `uniqueIdentifier` on its behalf.

The System 1 process is now ready to call `PXIMC_requestWindowLogicalAsServer`. The interface returned by `PXIMC_findInterfaces` can be used to request a memory window on that interface. This request is made by calling `PXIMC_requestWindowLogicalAsServer`. The parameters passed to `PXIMC_requestWindowLogicalAsServer` determine the exact behaviors of the window request, as described above. System 1 calls `PXIMC_requestWindowLogicalAsServer` using the values chosen above.

The process likely should call `PXIMC_waitForConnection` after requesting its window. If `PXIMC_waitForConnection` returns with status `PXIMC_SUCCESS`, the session has been successfully paired. This means that the local request matched a remote request, and that the two sessions were paired so

that they can communicate. PXIMC_waitForConnection returns addresses mapped to user-mode process addresses that the process can use to communicate with the remote process, using the rules to interpret the data that are dictated by the protocol used to initiate the connection.

The System 0 process can use the interfaceID returned from PXIMC_findInterfaces to call PXIMC_findWindows. It can use the list of windowIDs returned to pass them each to PXIMC_queryWindowInformation querying attribute PXIMC_WINDOW_CONNECTION_TYPE to determine if any server connections are available on the interface, and if so, if the System 0 process is compatible with them. The likely way the System 0 process determines compatibility is through protocolNumber being run on the server session, and also the windowData that the server session initialized its connection with. In this example, the System 0 process is looking for a windowData of "ABC Vendor Digitizer source". It will find that window using PXIMC_findWindows after the System 1 process has successfully called PXIMC_requestWindowLogicalAsServer as described above. System 0 can then use the window ID assigned to that session, also returned by PXIMC_findWindows, to connect specifically to associated session on System 1 by passing the windowID as the uniqueIdentifier value when requesting it's window.

System 0 can use all of the attribute values directly from PXIMC_queryWindowInformation. In this example, it would set zero for both minRemoteSize and maxRemoteSize, and set minLocalSize and maxLocalSize to the values from the remote window that it receives when querying the PXIMC_WINDOW_MIN_REMOTE_SIZE and PXIMC_WINDOW_MAX_REMOTE_SIZE attributes from PXIMC_queryWindowInformation. The window connection type is Logical, and the window connection type is Client, the protocolNumber must be the same number used on System 1, and the windowData can be set to NULL. The System 0 process then calls PXIMC_requestWindowLogicalAsClient with these parameter values.

When the System 0 process calls PXIMC_requestWindowLogicalAsClient, the session pairing algorithm is executed. As the System 0 request will be paired with the System 1 request, System 1's call to PXIMC_waitForConnection will now return. Assuming System 0 also calls PXIMC_waitForConnection immediately after requesting its window, it would also return. The System 0 process now has the address, the mappedLocalAddress returned from PXIMC_waitForConnection, that the System 1 digitizer will write data to.

Up to this point, this connection initialization has been equivalent to establishing a process-to-process connection.

To allow the System 1 digitizer to source data to the System 0 process, the System 1 process must now call PXIMC_getPhysicalAddress. This function returns the physical address that corresponds to the remote window. The System 1 process must now communicate with the digitizer to configure it with the physicalAddress that the digitizer can perform writes to. This configuration of the digitizer is specific to the vendor of the hardware, and is not covered by this specification.

A sequence diagram depicting the connection initialization is provided in Figure A-7.

The System 1 process has completed its initialization of the connection. The only other role the System 1 process has is to clean up the connection when the connection is terminated. The System 1 process can call PXIMC_waitForSessionEvent with a long timeout, and when it receives an PXIMC_EVENT_CONNECTION_CLOSED, it should call PXIMC_closeWindow.

Once the System 1 digitizer is programmed with the physical address, it can perform writes to the physical address. Any writes to the physical address will be received by the PXImc interface, which will write the data to the mappedLocalAddress for the System 0 process to read.

The System 1 digitizer has no way to directly assert an event or cause an interrupt on System 0. The System 1 process can call PXIMC_assertEvent at any time to cause the System 0 process to return from PXIMC_waitForSessionEvent.

After System 0 has completed communication with the System 1 digitizer, it calls PXIMC_closeWindow. This will cause the System 1 process PXIMC_waitForSessionEvent call to return, when it can then also call PXIMC_closeWindow to complete the PXIMC_cleanup of the connection. An example of connection termination is shown in Figure A-4.

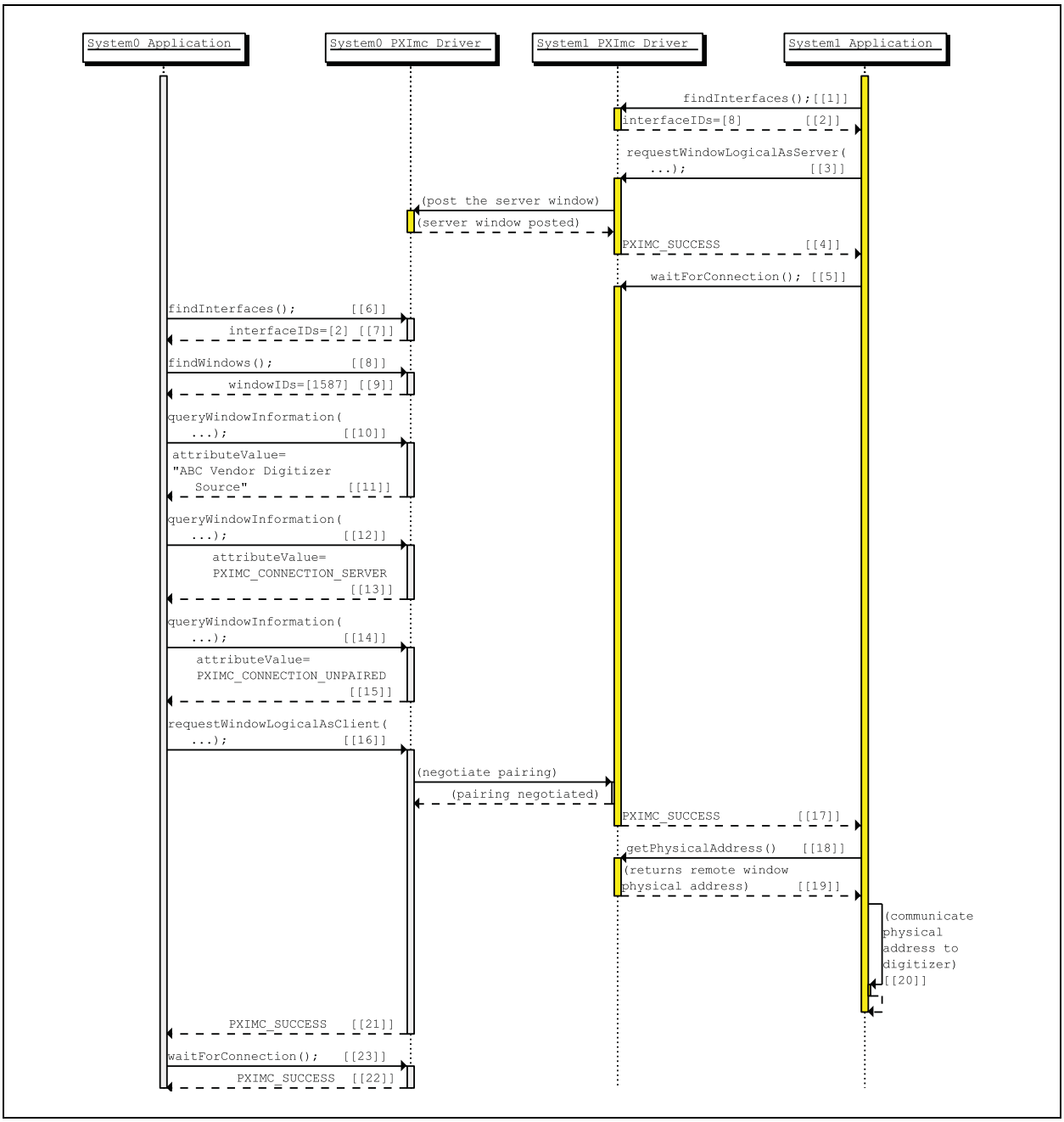


**Figure A-6.**Hardware to Process Connection Initiation

**Table A-5.**Figure A-6 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_findInterfaces (`<br>`  100, //maxNumberOfInterfaces`<br>`  interfaceIDsOut, //interfaceIDs`<br>`  numInterfacesOut`<br>`  //actualNumberOfInterfaces`<br>`  );` | The System 1 Application calls `findInterfaces()`. The application allocated an array of 100 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[2]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 8<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘8’. |
| [[3]] | `PXIMC_requestWindowLogicalAsServer (`<br>`  8, //interfaceID,`<br>`  0xABCD1000, //protocolNumber`<br>`  0, // maxLocalSize`<br>`  0, // minLocalSize`<br>`  0x1000, // maxRemoteSize`<br>`  0x400, // minRemoteSize`<br>`  0, // uniqueIdentifier`<br>`  “ABC Vendor Digitizer Source”,`<br>`  // windowData`<br>`  27, // windowDataSize`<br>`  sessionNumberOut //sessionNumber`<br>`  );` | System 1 requests its server window. It requests this window against interface ID ‘8’, as that is the interface found with `findInterfaces`. It passes the value ‘27’ for the `windowDataSize` because its `windowData` is 27 bytes long. |
| [[4]] | {`PXIMC_requestWindowLogicalAsServer` returns}<br>`sessionNumberOut` is set to 15<br>return value is `PXIMC_SUCCESS` | The window request is successful. The `sessionNumber` ‘15’ corresponds with this window request. The server window request has been posted to the remote system. |
| [[5]] | `PXIMC_waitForConnection (`<br>`  15, // sessionNumber`<br>`  0xFFFFFFFF, // timeoutInMilliseconds`<br>`  remoteAddressOut, // mappedRemoteAddress`<br>`  remoteSizeOut, // remoteSizeInBytes`<br>`  localAddressOut, // mappedLocalAddress`<br>`  localSizeOut // localSizeInBytes`<br>`  );` | System 1 waits until its window request is paired with a compatible window request on the remote system. |

**Table A-5.**Figure A-6 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[6]] | `PXIMC_findInterfaces (`<br>`  20, //maxNumberOfInterfaces`<br>`  interfaceIDsOut, //interfaceIDs`<br>`  numInterfacesOut`<br>`  //actualNumberOfInterfaces`<br>`  );` | The System 0 Application calls `findInterfaces()`. The application allocated an array of 20 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[7]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 2<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is '2'. |
| [[8]] | `PXIMC_findWindows (`<br>`  2, // interfaceID`<br>`  100, // maxNumberOfWindowIDs`<br>`  windowIDsOut, // windowIDs,`<br>`  numWindowsOut // actualNumberOfWindowIDs`<br>`  );` | System 0 is locating windows present on the remote side of interface '2'. The application allocated an array of 100 U32s in `windowIDsOut`, and passed the array size in `maxNumberOfWindowIDs`. |
| [[9]] | {`PXIMC_findWindows` returns}<br>`windowIDsOut` is set to 1587<br>`numWindowsOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One window was found. Its window ID is '1587'. |
| [[10]] | `PXIMC_queryWindowInformation (`<br>`  2, // interfaceID`<br>`  1587, // windowID`<br>`  PXIMC_U8_WINDOW_DATA, // attributeID`<br>`  1024, // maxSizeOfAttributeValue`<br>`  windowDataOut, // attributeValue`<br>`  sizeOut,`<br>`  //actualSizeOfAttributeValue`<br>`  );` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the window data of this window. The application allocated a 1024-byte buffer at `windowDataOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[11]] | {`PXIMC_queryWindowInformation` returns}<br>`windowDataOut` is set to "ABC Vendor Digitizer Source"<br>`sizeOut` is set to 27<br>return value is `PXIMC_SUCCESS` | The address `windowDataOut` now contains the `windowData`. The `sizeOut` value means that the `windowData` is 27 bytes long. |

**Table A-5.**Figure A-6 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[12]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`1587, // windowID`<br>`PXIMC_U32_WINDOW_CONNECTION_TYPE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`connectionTypeOut, // attributeValue`<br>`sizeOut, //actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the connection type of this window. The application allocated a single U32, 4 bytes, at `connectionTypeOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[13]] | {`PXIMC_queryWindowInformation` returns}<br>`connectionTypeOut` is set to `PXIMC_CONNECTION_SERVER`<br>sizeOut is set to 4<br>return Value is `PXIMC_SUCCESS` | `connectionTypeOut` now contains the connection type value. |
| [[14]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`1587, // windowID`<br>`PXIMC_U8_WINDOW_PAIRING_STATE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`pairingStateOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 0 is getting information about `windowID` '1587' on interface '2'. It is getting the pairing state of this window. The application allocated a single U32, 4 bytes, at `pairingStateOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[15]] | {`PXIMC_queryWindowInformation` returns}<br>`pairingStateOut` is set to `PXIMC_WINDOW_UNPAIRED`<br>`sizeOut` is set to 4<br>return Value is `PXIMC_SUCCESS` | `pairingStateOut` now contains the connection type value. |

**Table A-5.**Figure A-6 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[16]] | `PXIMC_requestWindowLogicalAsClient (`<br>`2, //interfaceID,`<br>`0xABCD1000, //protocolNumber`<br>`0x1000, // maxLocalSize`<br>`0x400, // minLocalSize`<br>`0, // maxRemoteSize`<br>`0, // minRemoteSize`<br>`1587, // uniqueIdentifier`<br>`sessionNumberOut //sessionNumber`<br>`);` | After examining the attributes of window '1587', System 0 wants to attempt to connect to it. It requests a window against interface 2, window 1587. |
| [[17]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to process memory-mapped pointer of the remote window<br>`remoteSizeOut` is set to set to the actual size of the remote window<br>`localAddressOut` is set to process memory-mapped pointer of the local window<br>`localSizeOut` is set to the actual size of the local window<br>return Value is `PXIMC_SUCCESS` | System 1's connection has been established once System 0 requested a compatible window, and the PXImc interface paired the System 0 and System 1 window requests. System 1 gets two pointers back, `remoteAddressOut` and `localAddressOut`, that it can use to access the local and remote window. |
| [[18]] | `PXIMC_getPhysicalAddress (`<br>`15, //sessionNumber`<br>`physicalAddressOut //physicalAddress`<br>`);` | System 1 calls `getPhysicalAddress` to get the physical address to which the digitizer will directly source its data. |
| [[19]] | {`PXIMC_getPhysicalAddress` returns}<br>`physicalAddressOut` is set to the physical address of the remote window<br>return Value is `PXIMC_SUCCESS` | System 1 now has the physical address to provide to the digitizer. |
| [[20]] | {System 1 Application provides `physicalAddressOut` to the digitizer} | The means for the System 1 Application to communicate the physical address to the digitizer is beyond the scope of this specification. |
| [[21]] | {`PXIMC_requestWindowLogicalAsClient` returns}<br>`sessionNumberOut` is set to 4<br>return Value is `PXIMC_SUCCESS` | The window request on System 0 is successful. The `sessionNumber` '4' corresponds with this window request. The client window request has been paired with a compatible server. |

**Table A-5.**Figure A-6 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[22]] | `PXIMC_waitForConnection (`<br>`4, // sessionNumber`<br>`0x0, //timeoutInMilliseconds`<br>`remoteAddressOut,// mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 0 waits until its window request is paired with a compatible window request on the remote system. Because its client window request was successful, it knows it has already been paired with a compatible server, but needs to call `waitForConnection` to get the pointers to the local and remote windows. |
| [[23]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to NULL, as there is no remote window<br>`remoteSizeOut` is set to set to zero, as there is no remote window<br>`localAddressOut` is set to process memory-mapped pointer of the local window<br>`localSizeOut` is set to the actual size of the local window<br>return Value is `PXIMC_SUCCESS` | System 0's `waitForConnection` returns. System 0 gets `localAddressOut` that it can use to access the local window. Any data written by the digitizer to the physical address provided to it in [[20]] will be available at `localAddressOut`. |

# A.4 Hardware sourcing data directly to hardware

Using the configuration shown in Figure A-1, the DAQ card on System 0 may want to receive data directly from the digitizer present in System 1 over PXImc. A common scenario where this type of connection is desired is if an output device is directly replaying data received by an input device. This section describes a common way this connection can be established.

While there are no processes directly participating in data transactions, a process is necessary on both System 0 and System 1 to establish the connection. After the connection is established, both the System 0 and System 1 processes are idle. The PXImc connection between System 0 and System 1 is represented by a unique interface. The interface between the two systems is returned on each system to the process by calling `PXIMC_findInterfaces`.

The DAQ card on System 0 must have the same interpretation of the data as the digitizer in System 1. As the DAQ card's interpretation of data is fixed by its register map, the System 0 digitizer must be able to source the data in a way that DAQ card can interpret it. The interpretation of the data is indicated by the protocol used to initiate the connection. To establish a connection, both processes must use the same `protocolNumber`.

The processes can initialize a connection by using a client/server relationship. The System 0 process must act as the "server", being the first to initiate the connection. The System 1 process must then act as the "client", initiating the connection after the server has initiated its portion of the connection. The PXImc

implementation will select the hardware resources that best meet the requirements of the connection. The physical resources used are determined based on the parameters of the connection request. System 0 will initiate the connection as the "server", and System 1 as the "client".

The System 0 process must determine what its requirements are for the local window. In this case, hardware on System 1 needs to write data to hardware on System 0. Therefore the System 0 process will request a local window where `localSize` is the local window size that the hardware is able to use. System 0 would set the `localSize` to the DAQ card base address register size.

This connection must use Physical window connection type because the destination is a physical address that is being manually supplied. The `physicalAddress` parameter must be the physical address that System 1 is being enabled to write to, in this case the physical address of the DAQ card base address register.

The `windowData` parameter is used to describe characteristics of the local process to the remote system. The data passed in the `windowData` is completely up to the process. One example of data that could be passed in the `windowData` is a string that describes the function of the process. The data provided in the `windowData` can be read on the remote system by calling `PXIMC_queryWindowInformation`. In this example, System 0 might pass "ABC Vendor DAQ Card 123" for the `windowData`. The System 0 process may choose to set the `uniqueIdentifier` to zero, and allow the PXImc interface to select a `uniqueIdentifier` on its behalf.

The System 0 process is now ready to call `PXIMC_requestWindowPhysicalAsServer`. The interface returned by `PXIMC_findInterfaces` can be used to request a memory window on that interface. This request is made by calling `PXIMC_requestWindowPhysicalAsServer`. The parameters passed to `PXIMC_requestWindowPhysicalAsServer` determine the exact behaviors of `PXIMC_requestWindowPhysicalAsServer`, as described above. System 0 calls `PXIMC_requestWindowPhysicalAsServer` using the values chosen above.

The process likely should call `PXIMC_waitForConnection` after requesting its window. If `PXIMC_waitForConnection` returns with status `PXIMC_SUCCESS`, the session has been successfully paired. This means that the local request matched a remote request, and that the two sessions were paired so that they can communicate.

The System 1 process can use the `interfaceID` returned from `PXIMC_findInterfaces` to call `PXIMC_findWindows`, and get the attributes of each window by calling `PXIMC_queryWindowInformation`. It can use the attribute values from `PXIMC_queryWindowInformation` to determine if any server connections are available on the interface, and if so, if the System 1 process is compatible with them. The likely way the System 1 process determines compatibility is through `protocolNumber` being run on the server session, and also the `windowData` that the server session initialized its connection with. In this example, the System 1 process is looking for a `windowData` of "ABC Vendor DAQ Card 123". It will find that window after the System 0 process has successfully called `PXIMC_requestWindowPhysicalAsServer` as described above. System 1 can then use the window ID returned by `PXIMC_findWindows` for the `uniqueIdentifier` parameter of the window request, to connect specifically to the session on System 0.

System 1 can use all of the attribute values directly returned by `PXIMC_queryWindowInformation` for its window request. In this example, it would set `minRemoteSize` and `maxRemoteSize` to the values from the remote window that it receives when querying the `PXIMC_WINDOW_MIN_LOCAL_SIZE` and `PXIMC_WINDOW_MAX_LOCAL_SIZE` attributes from `PXIMC_queryWindowInformation`. The window

location type is Physical and the window connection type is Client. The `protocolNumber` must be the same number used on System 0, and the `windowData` can be set to NULL. The System 1 process then calls `PXIMC_requestWindowPhysicalAsClient` with these parameter values.

When the System 1 process calls `PXIMC_requestWindowPhysicalAsClient`, the session pairing algorithm is executed. As the System 1 request will be paired with the System 0 request, System 0's call to `PXIMC_waitForConnection` will now return. Assuming System 1 also calls `PXIMC_waitForConnection` immediately after requesting its window, it would also return.

The System 0 process has now completed its initialization of the connection. System 1 can now perform accesses to the System 0 DAQ card.

Up to this point, this connection initialization has been equivalent to establishing a process-to-hardware connection.

To allow the System 1 digitizer to source data to the System 0 DAQ card, the System 1 process must now call `PXIMC_getPhysicalAddress`. This function returns the physical address that corresponds to the remote window. The System 1 process must now communicate with the digitizer to configure it with the `physicalAddress` that the digitizer can perform writes to. This configuration of the digitizer is specific to the vendor of the hardware, and is not covered by this specification.

A sequence diagram depicting the connection initialization is provided in Figure 7-8.

Once the System 1 digitizer is programmed with the physical address, it can perform writes to the physical address. Any writes to the physical address will be received by the PXImc interface, which will write the data to the `physicalAddress` provided on System 0, which in this example was the base address register address of the DAQ card.

The System 1 digitizer has no way to directly assert an event or cause an interrupt on System 0. The System 1 process can call `PXIMC_assertEvent` at any time to cause the System 0 process to return from `PXIMC_waitForSessionEvent`.

After use of the connection is complete, one of the processes must call `PXIMC_closeWindow` to start connection `PXIMC_cleanup`. The other System's process is notified of this by receiving an `PXIMC_EVENT_CONNECTION_CLOSED` event. That process should then also call `PXIMC_closeWindow`. At this point the connection has been terminated, and the resources used by the connection are freed. An example of connection termination is shown in Figure A-4.

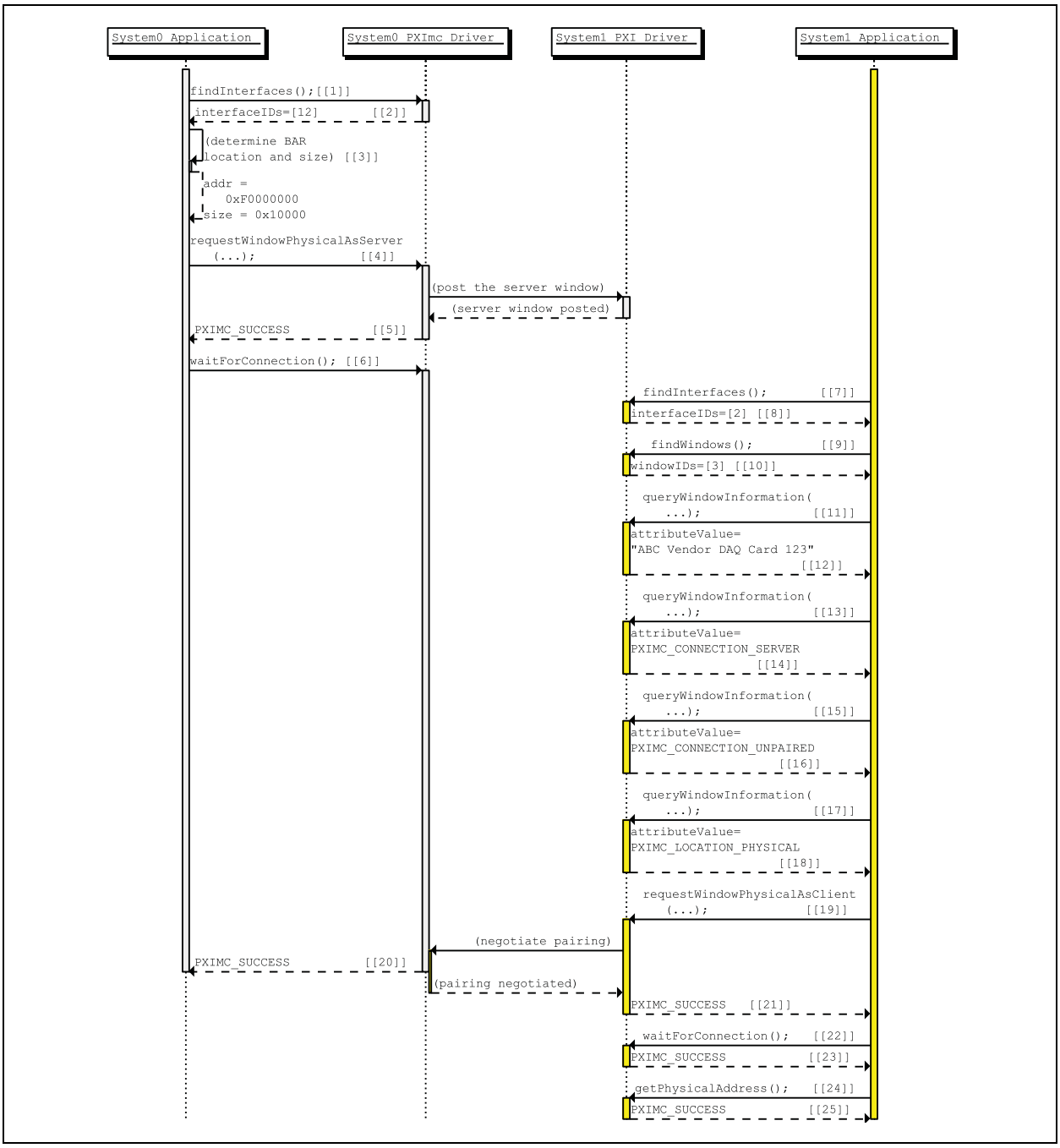


**Figure A-7.**Hardware to Hardware Connection Initiation

**Table A-6.**Figure A-7 Footnotes

| Footnote | Code | Explanation |
|---|---|---|
| [[1]] | `PXIMC_findInterfaces (`<br>`100, //maxNumberOfInterfaces`<br>`interfaceIDsOut, //interfaceIDs`<br>`numInterfacesOut`<br>`//actualNumberOfInterfaces`<br>`);` | The System 0 Application calls `findInterfaces()`. The application allocated an array of 100 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[2]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 12<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘12’. |
| [[3]] | {System 0 Application determines BAR address and size of the DAQ card}<br>BAR Address = 0x00000000F0000000<br>BAR Size = 0x10000 | The means for the System 0 Application to determine the address of the DAQ BAR is beyond the scope of this specification. |
| [[4]] | `PXIMC_requestWindowPhysicalAsServer (`<br>`12, //interfaceID,`<br>`0xABCD1000, //protocolNumber`<br>`0x10000, // localSize`<br>`0, // uniqueIdentifier`<br>`0x00000000F0000000, //physicalAddress`<br>`“ABC Vendor DAQ Card 123”,`<br>`// windowData`<br>`23, // windowDataSize`<br>`sessionNumberOut //sessionNumber`<br>`);` | System 0 requests its server window. It requests this window against interface ID ‘12’, as that is the interface found with `findInterfaces`. It passes the value ‘23’ for the `windowDataSize` because its `windowData` is 23 bytes long. |
| [[5]] | {`PXIMC_requestWindowLogicalAsServer` returns}<br>`sessionNumberOut` is set to 15<br>return value is `PXIMC_SUCCESS` | The window request is successful. The `sessionNumber` ‘15’ corresponds with this window request. The server window request has been posted to the remote system. |

**Table A-6.**Figure A-7 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[6]] | `PXIMC_waitForConnection (`<br>`15, // sessionNumber`<br>`0xFFFFFFFF, // timeoutInMilliseconds`<br>`remoteAddressOut, // mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 0 waits until its window request is paired with a compatible window request on the remote system. |
| [[7]] | `PXIMC_findInterfaces (`<br>`20, //maxNumberOfInterfaces`<br>`interfaceIDsOut, //interfaceIDs`<br>`numInterfacesOut`<br>`//actualNumberOfInterfaces`<br>`);` | The System 1 Application calls `findInterfaces()`. The application allocated an array of 20 U32s in `interfacesIDsOut`, and passed the array size in `maxNumberOfInterfaces`. |
| [[8]] | {`PXIMC_findInterfaces` returns}<br>`interfaceIDsOut` is set to 2<br>`numInterfacesOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One interface was found. Its interface ID is ‘2’. |
| [[9]] | `PXIMC_findWindows (`<br>`2, // interfaceID`<br>`100, // maxNumberOfWindowIDs`<br>`windowIDsOut, // windowIDs,`<br>`numWindowsOut // actualNumberOfWindowIDs`<br>`);` | System 1 is locating windows present on the remote side of interface ‘2’. The application allocated an array of 100 U32s in `windowIDsOut`, and passed the array size in `maxNumberOfWindowIDs`. |
| [[10]] | {`PXIMC_findWindows` returns}<br>`windowIDsOut` is set to 3<br>`numWindowsOut` is set to 1<br>return value is `PXIMC_SUCCESS` | One window was found. Its window ID is ‘3’. |

**Table A-6.**Figure A-7 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[11]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U8_WINDOW_DATA, // attributeID`<br>`1024, // maxSizeOfAttributeValue`<br>`windowDataOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 1 is getting information about `windowID` '3' on interface '2'. It is getting the window data of this window. The application allocated a 1024-byte buffer at `windowDataOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[12]] | {`PXIMC_queryWindowInformation` returns}<br>`windowDataOut` is set to "ABC Vendor DAQ Card 123"<br>`sizeOut` is set to 23<br>return value is `PXIMC_SUCCESS` | The address `windowDataOut` now contains the `windowData`. The `sizeOut` value means that the `windowData` is 23 bytes long. |
| [[13]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U32_WINDOW_CONNECTION_TYPE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`connectionTypeOut, // attributeValue`<br>`sizeOut, //actualSizeOfAttributeValue`<br>`);` | System 1 is getting information about `windowID` '3' on interface '2'. It is getting the connection type of this window. The application allocated a single U32, 4 bytes, at `connectionTypeOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |
| [[14]] | {`PXIMC_queryWindowInformation` returns}<br>`connectionTypeOut` is set to `PXIMC_CONNECTION_SERVER`<br>`sizeOut` is set to 4<br>return value is `PXIMC_SUCCESS` | `connectionTypeOut` now contains the connection type value. |
| [[15]] | `PXIMC_queryWindowInformation (`<br>`2, // interfaceID`<br>`3, // windowID`<br>`PXIMC_U8_WINDOW_PAIRING_STATE,`<br>`// attributeID`<br>`4, // maxSizeOfAttributeValue`<br>`pairingStateOut, // attributeValue`<br>`sizeOut,`<br>`//actualSizeOfAttributeValue`<br>`);` | System 1 is getting information about `windowID` '3' on interface '2'. It is getting the pairing state of this window. The application allocated a single U32, 4 bytes, at `pairingStateOut`, and is providing the size of the buffer in `maxSizeOfAttributeValue`. |

**Table A-6.**Figure A-7 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[16]] | {PXIMC_queryWindowInformation returns}<br>pairingStateOut is set to PXIMC_WINDOW_UNPAIRED<br>sizeOut is set to 4<br>return value is PXIMC_SUCCESS | pairingStateOut now contains the connection type value. |
| [[17]] | PXIMC_queryWindowInformation (<br>2, // interfaceID<br>3, // windowID<br>PXIMC_U8_WINDOW_LOCATION_TYPE,<br>// attributeID<br>4, // maxSizeOfAttributeValue<br>pairingStateOut, // attributeValue<br>sizeOut,<br>//actualSizeOfAttributeValue<br>); | System 1 is getting information about windowID '3' on interface '2'. It is getting the location of this window. The application allocated a single U32, 4 bytes, at pairingStateOut, and is providing the size of the buffer in maxSizeOfAttributeValue. |
| [[18]] | {PXIMC_queryWindowInformation returns}<br>pairingStateOut is set to PXIMC_LOCATION_PHYSICAL<br>sizeOut is set to 4<br>return value is PXIMC_SUCCESS | pairingStateOut now contains the connection type value. |
| [[19]] | PXIMC_requestWindowPhysicalAsClient (<br>2, //interfaceID,<br>0xABCD1000, //protocolNumber<br>0x10000, // maxRemoteSize<br>0, // minRemoteSize<br>3, // uniqueIdentifier<br>sessionNumberOut //sessionNumber<br>); | After examining the attributes of window '3', System 1 wants to attempt to connect to it. It requests a window against interface 2, window 3. |
| [[20]] | {PXIMC_waitForConnection returns}<br>remoteAddressOut is set to NULL, as there is no remote window<br>remoteSizeOut is set to set to zero, as there is no remote window<br>localAddressOut is set to NULL, as the process cannot access the local window<br>localSizeOut is set to zero, as the process cannot access the local window<br>return value is PXIMC_SUCCESS | System 0's connection has been established once System 1 requested a compatible window, and the PXImc interface paired the System 0 and System 1 window requests. System 0 cannot use its window to perform any I/O, but it has enabled System 1 to access the DAQ BAR. |

**Table A-6.**Figure A-7 Footnotes (Continued)

| Footnote | Code | Explanation |
|---|---|---|
| [[21]] | {`PXIMC_requestWindowLogicalAsClient` returns}<br>`sessionNumberOut` is set to 4<br>return value is `PXIMC_SUCCESS` | The window request on System 1 is successful. The `sessionNumber` '4' corresponds with this window request. The client window request has been paired with a compatible server. |
| [[22]] | `PXIMC_waitForConnection (`<br>`4, // sessionNumber`<br>`0x0, //timeoutInMilliseconds`<br>`remoteAddressOut,// mappedRemoteAddress`<br>`remoteSizeOut, // remoteSizeInBytes`<br>`localAddressOut, // mappedLocalAddress`<br>`localSizeOut // localSizeInBytes`<br>`);` | System 1 waits until its window request is paired with a compatible window request on the remote system. Because its client window request was successful, it knows it has already been paired with a compatible server, but needs to call `waitForConnection` to get the pointers to the remote window. |
| [[23]] | {`PXIMC_waitForConnection` returns}<br>`remoteAddressOut` is set to process memory-mapped pointer of the remote window (the DAQ BAR)<br>`remoteSizeOut` is set to set to the actual size of the remote window (the exact size of the DAQ BAR)<br>`localAddressOut` is set to NULL, as there is no local window<br>`localSizeOut` is set to zero, as there is no local window<br>return value is `PXIMC_SUCCESS` | System 1's `waitForConnection` returns. System 0 gets two pointers back, `remoteAddressOut` and `localAddressOut`, that it can use to access the local and remote window. `localAddressOut` will be NULL and `localSizeOut` will be zero, as there is no local window. When System 1 accesses `remoteAddressOut`, the result will be that the DAQ BAR in System 1 is accessed. |
| [[24]] | `PXIMC_getPhysicalAddress (`<br>`4, //sessionNumber`<br>`physicalAddressOut //physicalAddress`<br>`);` | System 1 calls `getPhysicalAddress` to get the physical address to which the digitizer will directly source its data. |
| [[25]] | {`PXIMC_getPhysicalAddress` returns}<br>`physicalAddressOut` is set to the physical address of the remote window<br>return value is `PXIMC_SUCCESS` | System 1 now has the physical address to provide to the digitizer. The means for the System 1 Application to communicate the physical address to the digitizer is beyond the scope of this specification. Once the digitizer is provided with this physical address, any data written by the digitizer to the physical address will be received by the DAQ card on System 0. |

## A.5 Process sourcing data to hardware and hardware sourcing data to process

Using the configuration shown in Figure A-1, a process on System 0 may want to send data directly to the digitizer present in System 1 over PXImc. At the same time, the process may want to receive data directly from the digitizer. A common scenario where this type of connection is desired is if a process is directly configuring an input device, and at the same time the input device is streaming its input data directly to the process. This section describes how this connection can be established.

This connection should be viewed as two separate connections—one connection for the System 0 process to source data to the System 1 digitizer, and another connection enabling the digitizer to write data to the System 0 process. Follow the examples in Section 0 and Section A.3 to set up these two independent connections.

## A.6 Bi-directional link where hardware is sourcing data directly to hardware in both directions

Using the configuration shown in Figure A-1, the DAQ card in System 0 may want to write data directly the digitizer in System 1, and at the same time the digitizer may want to write data directly to the DAQ card in System 0. A common scenario where this type of connection is desired is if one card is controlling the input device's control registers, and at the same time the input device is streaming it's input data directly to the other hardware. This section describes how this connection can be established.

This connection should be viewed as two separate connections—one enabling the System 1 digitizer to write to the System 0 DAQ card, and an independent connection enabling the System 0 DAQ card to write to the System 1 digitizer. Follow the example in Section A.4 two times, inverting the roles of which system acts as the "server" and which acts as the "client" after executing the example once.

# B. Appendix: pximc.h

```
#if !defined (__pximc_h__)
#define __pximc_h__

/*-----------------------------------------------------------------------*/
/* Distributed by PXISA                                                  */
/* Do not modify the contents of this file.                              */
/*-----------------------------------------------------------------------*/
/*                                                                       */
/* Title   : pximc.h                                                     */
/* Date    : 07-30-2009                                                  */
/* Purpose : Definitions for using the PXImc API, compliant with         */
/*           revision 1.0 of the PXImc software specification            */
/*                                                                       */
/*-----------------------------------------------------------------------*/

#if defined(__cplusplus) || defined(__cplusplus__)
   extern "C" {
#endif

/*- PXIMC Types -------------------------------------------------------*/


#if defined(_WIN64) || ((defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
|| defined(__NT__)) && !defined(_NI_mswin16_))
#if (defined(_MSC_VER) && (_MSC_VER >= 1200)) || (defined(_CVI_) && (_CVI_ >=
700)) || (defined(__BORLANDC__) && (__BORLANDC__ >= 0x0520))

#if ((defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__))
&& !defined(_NI_mswin16_))
#define _PXIMC_FUNC        _stdcall
#elif defined (WIN64)
#define _PXIMC_FUNC
#else
#endif

typedef __int8           int8_t;
typedef unsigned __int8  uint8_t;
typedef __int16          int16_t;
typedef unsigned __int16 uint16_t;
typedef __int32          int32_t;
typedef unsigned __int32 uint32_t;
typedef __int64          int64_t;
typedef unsigned __int64 uint64_t;
```

```
#endif
#elif defined(__GNUC__) && (__GNUC__ >= 3)
#define _PXIMC_FUNC
#include <limits.h>
#include <sys/types.h>
#include <stdint.h>
#else
/* This platform does not support 64-bit types */
#endif

#if !defined (UINT64_MAX)
#define UINT64_MAX  18446744073709551615ULL
#endif
#if !defined (UINT32_MAX)
#define UINT32_MAX  4294967295UL
#endif

typedef int32_t tPXIMC_Status;

/*-----------------------------------------------------------------------------*/
/*                                                                             */
/* PXIMC API function definitions                                              */
/*                                                                             */
/*-----------------------------------------------------------------------------*/

tPXIMC_Status _PXIMC_FUNC PXIMC_findInterfaces (
   uint32_t   maxNumberOfInterfaces,
   uint32_t * interfaceIDs,
   uint32_t * actualNumberOfInterfaces
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_queryInterfaceInformation (
   uint32_t   interfaceID,
   uint32_t   attributeID,
   uint32_t   maxSizeOfAttributeValue,
   void     * attributeValue,
   uint32_t * actualSizeOfAttributeValue
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_waitForInterfaceEvent (
   uint32_t   interfaceID,
   uint32_t   timeoutInMilliseconds,
   uint32_t * reasonCode
   );
```

```
tPXIMC_Status _PXIMC_FUNC PXIMC_findWindows (
   uint32_t   interfaceID,
   uint32_t   maxNumberOfWindowIDs,
   uint32_t * windowIDs,
   uint32_t * actualNumberOfWindowIDs
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_queryWindowInformation (
   uint32_t   interfaceID,
   uint32_t   windowID,
   uint32_t   attributeID,
   uint32_t   maxSizeOfAttributeValue,
   void     * attributeValue,
   uint32_t * actualSizeOfAttributeValue
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_requestWindowLogicalAsServer (
   uint32_t        interfaceID,
   uint32_t        protocolNumber,
   uint64_t        maxLocalSize,
   uint64_t        minLocalSize,
   uint64_t        maxRemoteSize,
   uint64_t        minRemoteSize,
   uint32_t        uniqueIdentifier,
   const uint8_t * windowData,
   uint32_t        windowDataSize,
   uint32_t      * sessionNumber
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_requestWindowLogicalAsClient (
   uint32_t   interfaceID,
   uint32_t   protocolNumber,
   uint64_t   maxLocalSize,
   uint64_t   minLocalSize,
   uint64_t   maxRemoteSize,
   uint64_t   minRemoteSize,
   uint32_t   uniqueIdentifier,
   uint32_t * sessionNumber
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_requestWindowLogicalAsPeer (
   uint32_t        interfaceID,
   uint32_t        protocolNumber,
   uint64_t        maxLocalSize,
   uint64_t        minLocalSize,
   uint64_t        maxRemoteSize,
```

```
    uint64_t        minRemoteSize,
    uint32_t        uniqueIdentifier,
    const uint8_t * windowData,
    uint32_t        windowDataSize,
    uint32_t      * sessionNumber
    );

tPXIMC_Status _PXIMC_FUNC PXIMC_requestWindowPhysicalAsServer (
    uint32_t        interfaceID,
    uint32_t        protocolNumber,
    uint64_t        localSize,
    uint32_t        uniqueIdentifier,
    uint64_t        physicalAddress,
    const uint8_t * windowData,
    uint32_t        windowDataSize,
    uint32_t      * sessionNumber
    );

tPXIMC_Status _PXIMC_FUNC PXIMC_requestWindowPhysicalAsClient (
    uint32_t   interfaceID,
    uint32_t   protocolNumber,
    uint64_t   maxRemoteSize,
    uint64_t   minRemoteSize,
    uint32_t   uniqueIdentifier,
    uint32_t * sessionNumber
    );

tPXIMC_Status _PXIMC_FUNC PXIMC_waitForConnection (
    uint32_t    sessionNumber,
    uint32_t    timeoutInMilliseconds,
    void     ** mappedRemoteAddress,
    uint64_t *  remoteSizeInBytes,
    void     ** mappedLocalAddress,
    uint64_t *  localSizeInBytes
    );

tPXIMC_Status _PXIMC_FUNC PXIMC_getPhysicalAddress (
    uint32_t   sessionNumber,
    uint64_t * physicalAddress
    );

tPXIMC_Status _PXIMC_FUNC PXIMC_enableDeviceAccess (
    uint32_t    sessionNumber,
    uint32_t    accessMode,
    uint32_t    deviceBusNumber,
    uint32_t    deviceDevNumber,
```

```
   uint32_t    deviceFuncNumber
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_assertEvent (
   uint32_t sessionNumber
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_waitForSessionEvent (
   uint32_t   sessionNumber,
   uint32_t   timeoutInMilliseconds,
   uint32_t * reasonCode
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_closeWindow (
   uint32_t sessionNumber
   );

tPXIMC_Status _PXIMC_FUNC PXIMC_cleanup ();

/*---------------------------------------------------------------------------*/
/*                                                                           */
/* PXIMC API function constants                                              */
/*                                                                           */
/*---------------------------------------------------------------------------*/

enum
{
   PXIMC_SPEC_VERSION                    = 0x00010000
};

enum
{
   _PXIMC_STR_BASE                       = 0x10000000UL,
   _PXIMC_U8_BASE                        = 0x20000000UL,
   _PXIMC_U32_BASE                       = 0x30000000UL,
   _PXIMC_U64_BASE                       = 0x40000000UL
};



/*---------------------------------------------------------------------------*/
/* PXIMC_queryInterfaceInformation                                           */
/*---------------------------------------------------------------------------*/

/*   PXIMC_queryInterfaceInformation attributes                              */
/*      string attributes for PXIMC_queryInterfaceInformation                */
```

```
enum ePXIMC_QueryInterfaceInformation_StrAttribute
{
   PXIMC_STR_MANF_NAME                    = (_PXIMC_STR_BASE + 1UL),
      /*0x10000001, 268435457 */
   PXIMC_STR_MODEL_NAME                   = (_PXIMC_STR_BASE + 2UL),
      /*0x10000002, 268435458 */
   PXIMC_STR_SERIAL_NUM                   = (_PXIMC_STR_BASE + 3UL),
      /*0x10000003, 268435459 */
   PXIMC_STR_LOG_DATA                     = (_PXIMC_STR_BASE + 4UL),
      /*0x10000004, 268435460 */
   PXIMC_STR_INTERFACE_NAME               = (_PXIMC_STR_BASE + 5UL),
      /*0x10000005, 268435461 */
   PXIMC_STR_REMOTE_OS                    = (_PXIMC_STR_BASE + 6UL)
      /*0x10000006, 268435462 */
};

/*      u32 attributes for PXIMC_queryInterfaceInformation       */
enum ePXIMC_QueryInterfaceInformation_U32Attribute
{
   PXIMC_U32_PROTOCOL_VERSION             = (_PXIMC_U32_BASE + 1UL),
      /*0x30000001, 805306369 */
   PXIMC_U32_MANF_ID                      = (_PXIMC_U32_BASE + 2UL),
      /*0x30000002, 805306370 */
   PXIMC_U32_INTERFACE_STATE              = (_PXIMC_U32_BASE + 3UL),
      /*0x30000003, 805306371 */
   PXIMC_U32_INTERFACE_DEVICE_ID          = (_PXIMC_U32_BASE + 4UL),
      /*0x30000004, 805306372 */
   PXIMC_U32_INTERFACE_VENDOR_ID          = (_PXIMC_U32_BASE + 5UL),
      /*0x30000005, 805306373 */
   PXIMC_U32_INTERFACE_SS_ID              = (_PXIMC_U32_BASE + 6UL),
      /*0x30000006, 805306374 */
   PXIMC_U32_INTERFACE_SS_VENDOR_ID       = (_PXIMC_U32_BASE + 7UL),
      /*0x30000007, 805306375 */
   PXIMC_U32_INTERFACE_BUS                = (_PXIMC_U32_BASE + 8UL),
      /*0x30000008, 805306376 */
   PXIMC_U32_INTERFACE_DEV                = (_PXIMC_U32_BASE + 9UL),
      /*0x30000009, 805306377 */
   PXIMC_U32_INTERFACE_FUNC               = (_PXIMC_U32_BASE + 10UL),
      /*0x3000000A, 805306378 */
   PXIMC_U32_INTERFACE_LOCAL              = (_PXIMC_U32_BASE + 11UL),
      /*0x3000000B, 805306379 */
   PXIMC_U32_REMOTE_ENDIANNESS            = (_PXIMC_U32_BASE + 12UL),
      /*0x3000000C, 805306380 */
   PXIMC_U32_REMOTE_WORD_SIZE             = (_PXIMC_U32_BASE + 13UL)
      /*0x3000000D, 805306381 */
};
```

```
/*   end PXIMC_queryInterfaceInformation attributes                          */

/*   PXIMC_U32_INTERFACE_STATE return values                                 */
enum ePXIMC_QueryInterfaceInformation_InterfaceState
{
   PXIMC_STATE_UP                          = 1,
   PXIMC_STATE_DOWN                        = 2
};

/*   PXIMC_U32_INTERFACE_LOCAL return values                                 */
enum ePXIMC_QueryInterfaceInformation_InterfaceLocal
{
   PXIMC_LOCAL                             = 1,
   PXIMC_REMOTE                            = 2
};
/*-------------------------------------------------------------------------*/
/* end PXIMC_queryInterfaceInformation                                     */
/*-------------------------------------------------------------------------*/



/* PXIMC_waitForInterfaceEvent                                             */
enum ePXIMC_WaitForInterfaceEvent_ReasonCode
{
   PXIMC_EVENT_INTERFACE_STATE_CHANGE      = 1,
   PXIMC_EVENT_WINDOW_STATE_CHANGE         = 2
};



/*-------------------------------------------------------------------------*/
/* PXIMC_queryWindowInformation                                            */
/*-------------------------------------------------------------------------*/

/*   PXIMC_queryWindowInformation attributes                                */
/*      u8 array attributes                                                 */
enum ePXIMC_QueryWindowInformation_U8Attribute
{
   PXIMC_U8_WINDOW_DATA                    = (_PXIMC_U8_BASE + 1UL)
      /*0x20000001, 536870913 */
};

/*      u32 attributes                                                  */
enum ePXIMC_QueryWindowInformation_U32Attribute
{
   PXIMC_U32_WINDOW_CONNECTION_TYPE        = (_PXIMC_U32_BASE + 1UL),
```

```
      /*0x30000001, 805306369 */
   PXIMC_U32_WINDOW_LOCATION_TYPE         = (_PXIMC_U32_BASE + 2UL),
      /*0x30000002, 805306370 */
   PXIMC_U32_WINDOW_PROTOCOL_NUMBER       = (_PXIMC_U32_BASE + 3UL),
      /*0x30000003, 805306371 */
   PXIMC_U32_WINDOW_PAIRING_STATE         = (_PXIMC_U32_BASE + 4UL),
      /*0x30000004, 805306372 */
   PXIMC_U32_SESSION_EVENT_STATUS         = (_PXIMC_U32_BASE + 5UL)
      /*0x30000005, 805306373 */
};

/*      u64 attributes                                              */
enum ePXIMC_QueryWindowInformation_U64Attribute
{
   PXIMC_U64_WINDOW_MIN_REMOTE_SIZE       = (_PXIMC_U64_BASE + 1UL),
      /*0x40000001, 1073741825 */
   PXIMC_U64_WINDOW_MAX_REMOTE_SIZE       = (_PXIMC_U64_BASE + 2UL),
      /*0x40000002, 1073741826 */
   PXIMC_U64_WINDOW_MIN_LOCAL_SIZE        = (_PXIMC_U64_BASE + 3UL),
      /*0x40000003, 1073741827 */
   PXIMC_U64_WINDOW_MAX_LOCAL_SIZE        = (_PXIMC_U64_BASE + 4UL)
      /*0x40000004, 1073741828 */
};
/*   end PXIMC_queryWindowInformation attributes                    */

/*   PXIMC_U32_WINDOW_CONNECTION_TYPE return values                 */
enum ePXIMC_QueryWindowInformation_WindowConnectionType
{
   PXIMC_CONNECTION_SERVER                = 1,
   PXIMC_CONNECTION_CLIENT                = 2,
   PXIMC_CONNECTION_PEER                  = 3
};

/*   PXIMC_U32_WINDOW_LOCATION_TYPE return values                   */
enum ePXIMC_QueryWindowInformation_WindowLocationType
{
   PXIMC_LOCATION_LOGICAL                 = 1,
   PXIMC_LOCATION_PHYSICAL                = 2
};

/*   PXIMC_U32_WINDOW_PAIRING_STATE return values                   */
enum ePXIMC_QueryWindowInformation_WindowPairingState
{
   PXIMC_WINDOW_PAIRED                    = 1,
   PXIMC_WINDOW_UNPAIRED                  = 2
};
```

```
/*   PXIMC_U32_WINDOW_EVENT_STATUS return values                          */
enum ePXIMC_QueryWindowInformation_WindowEventStatus
{
   PXIMC_WINDOW_REMOTE_EVENT_PENDING      = 1,
   PXIMC_WINDOW_REMOTE_SESSION_WAITING    = 2,
   PXIMC_WINDOW_LOCAL_EVENT_PENDING       = 4,
   PXIMC_WINDOW_LOCAL_SESSION_WAITING     = 8
};
/*--------------------------------------------------------------------------*/
/* end PXIMC_queryWindowInformation                                      */
/*--------------------------------------------------------------------------*/



/* PXIMC_requestWindow                                                   */
enum
{
   PXIMC_MAXIMUM_WINDOW_SIZE              = UINT64_MAX
};

enum
{
   PXIMC_TIMEOUT_INFINITE                 = UINT32_MAX
};

/*   PXIMC_enableDeviceAccess                                             */
enum ePXINTB_EnableDeviceAccess_AccessMode
{
   PXIMC_DEVICE_ACCESS_READ      = 1,
   PXIMC_DEVICE_ACCESS_WRITE     = 2,
   PXIMC_DEVICE_ACCESS_CLEAR_ALL = 0x80000000
};

/* PXIMC_waitForSessionEvent                                             */
enum ePXIMC_WaitForSessionEvent_ReasonCode
{
   PXIMC_EVENT_ASSERTED                   = 1,
   PXIMC_EVENT_CONNECTION_CLOSED          = 2,
   PXIMC_EVENT_INTERFACE_DOWN             = 3
};



/*--------------------------------------------------------------------------*/
/*                                                                          */
```

```
/* PXIMC API status values                                                    */
/*                                                                            */
/*----------------------------------------------------------------------------*/
enum
{
   _PXIMC_ERROR_BASE                    = (0x80000000L),
   _PXIMC_WARNING_BASE                  = 0x10000000UL
};

enum ePXIMC_Status
{
/* Success                                                                    */
   PXIMC_SUCCESS                        = 0,

/* Error                                                                      */
   PXIMC_INSUFFICIENT_SPACE             = _PXIMC_ERROR_BASE + 0x1000L,
      /*0x80001000,  -2147479552*/
   PXIMC_INVALID_INTERFACE              = _PXIMC_ERROR_BASE + 0x1001L,
      /*0x80001001,  -2147479551*/
   PXIMC_INTERFACE_DOWN                 = _PXIMC_ERROR_BASE + 0x1002L,
      /*0x80001002,  -2147479550*/
   PXIMC_NSUP_ATTRIBUTE                 = _PXIMC_ERROR_BASE + 0x1003L,
      /*0x80001003,  -2147479549*/
   PXIMC_INVALID_ARGUMENT               = _PXIMC_ERROR_BASE + 0x1004L,
      /*0x80001004,  -2147479548*/
   PXIMC_SPACE_NOT_AVAILABLE            = _PXIMC_ERROR_BASE + 0x1005L,
      /*0x80001005,  -2147479547*/
   PXIMC_UID_CONFLICT                   = _PXIMC_ERROR_BASE + 0x1006L,
      /*0x80001006,  -2147479546*/
   PXIMC_NO_PAIRING                     = _PXIMC_ERROR_BASE + 0x1007L,
      /*0x80001007,  -2147479545*/
   PXIMC_PHY_RESOURCE_NOT_AVAILABLE     = _PXIMC_ERROR_BASE + 0x1008L,
      /*0x80001008,  -2147479544*/
   PXIMC_INVALID_SESSION                = _PXIMC_ERROR_BASE + 0x1009L,
      /*0x80001009,  -2147479543*/
   PXIMC_NO_WINDOW                      = _PXIMC_ERROR_BASE + 0x100AL,
      /*0x8000100A,  -2147479542*/
   PXIMC_SESSION_CLOSED                 = _PXIMC_ERROR_BASE + 0x100BL,
      /*0x8000100B,  -2147479541*/
   PXIMC_INVALID_WINDOW                 = _PXIMC_ERROR_BASE + 0x100CL,
      /*0x8000100C,  -2147479540*/
   PXIMC_INVALID_RESOURCE               = _PXIMC_ERROR_BASE + 0x100DL,
      /*0x8000100D,  -2147479539*/
   PXIMC_ALIGNMENT_ERROR                = _PXIMC_ERROR_BASE + 0x100EL,
      /*0x8000100E,  -2147479538*/
```

```
/* Warning                                                                  */
   PXIMC_NO_PROVIDER                    = _PXIMC_WARNING_BASE + 0x1000UL,
      /*0x10001000, 268439552*/
   PXIMC_TIMEOUT                        = _PXIMC_WARNING_BASE + 0x1001UL,
      /*0x10001001, 268439553*/
};

#if defined(__cplusplus) || defined(__cplusplus__)
}
#endif

#endif  /*  #if !defined (__pximc_h__)  */

/*- The End -----------------------------------------------------------------*/
```

# C. Appendix: PXImc Background Information

This section includes some background information about Non-transparent bridges, PCI(e) BARs, BIOS operations, and device drivers. This information is intended only for reference for potential implementers of a PXImc Logic Block. This section contains no rules that are requirements of the PXImc Specification.

## C.1 PCI(e) BARs and the BIOS

Non-transparent bridges (NTBs) have one or more Base Address Register (BAR). The size of the BAR(s) is set based on hardware-specific configuration. Some BARs may have a fixed size, others may be configurable. Because of this hardware-specific behavior, both the size of the BARs and how the sizes of BARs are set aren't covered here; these details are left for the PXImc Logic Block Vendor to determine individually.

The BAR sizes of the NTB are fixed for a given boot of the system. It is customary for the system BIOS to enumerate all PCI(e) devices in the system, determine the memory and I/O resource needs of the devices, and assign a specific address to the BARs of every PCI(e) device. Once that process is complete, both the size and location of the BARs are set until the PCI(e) tree is re-initialized (usually at restart). After the PCI(e) tree initialization algorithm completes, an NTB is in the following state:

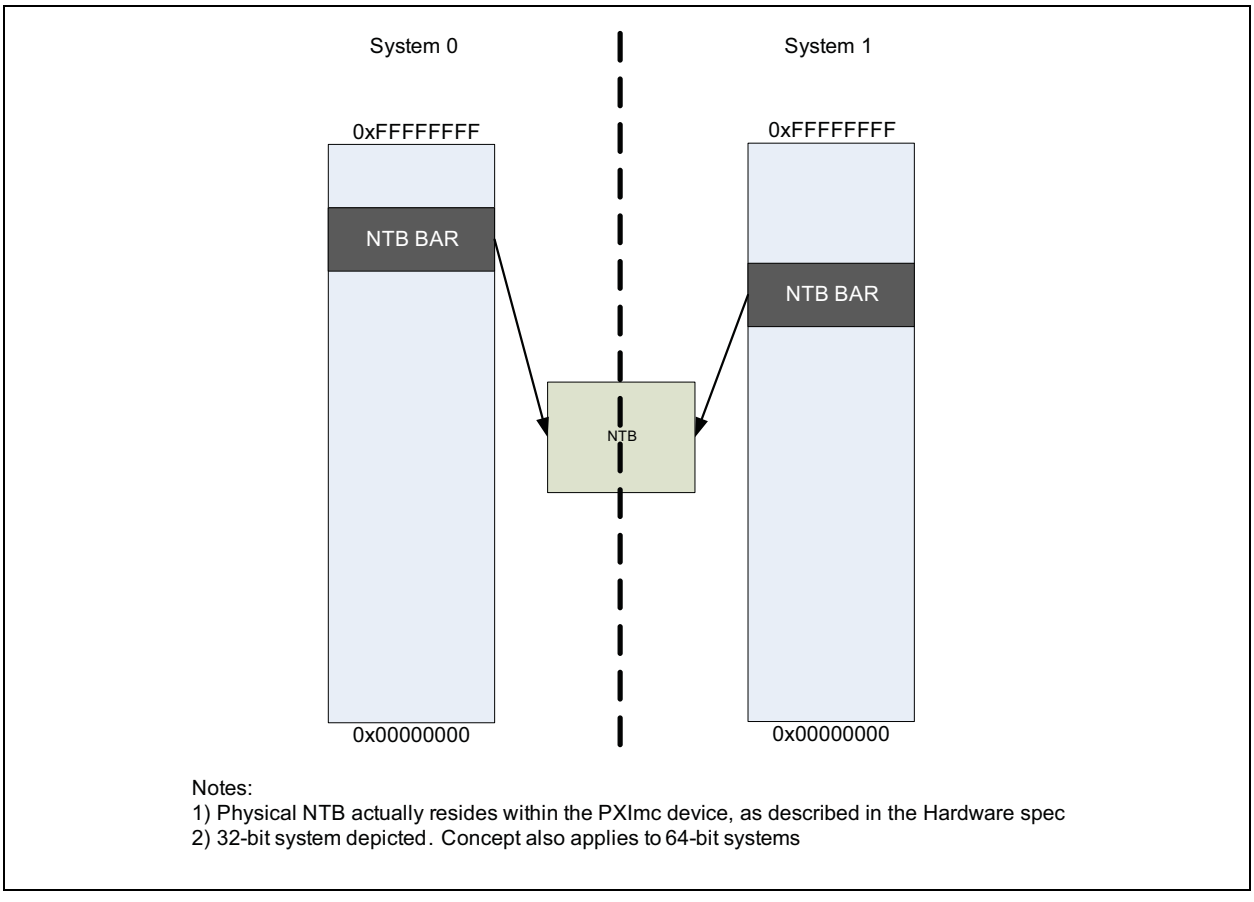


**Figure C-1.** PXImc Initialization Overview

## C.2 PCI(e) Device Drivers

The operating system attempts to load a device driver for every PCI(e) device in the system. The PXImc Logic Block Vendor must provide a device driver to get loaded by both systems' operating system. This device driver is responsible for understanding the behaviors and functionality of the hardware it represents. One function the NTB device driver needs to be able to accomplish is to be able to map accesses of the NTB BAR(s) to physical memory. The means for generating this mapping cannot be covered here, as each NTB may implement this differently (direct address translation vs. look up table translation, for example). The NTB driver must be able to create some memory mapping where BAR accesses get translated to physical memory accesses, and the NTB driver needs to own the physical memory that is targeted. The simplest depiction of this functionality is below.

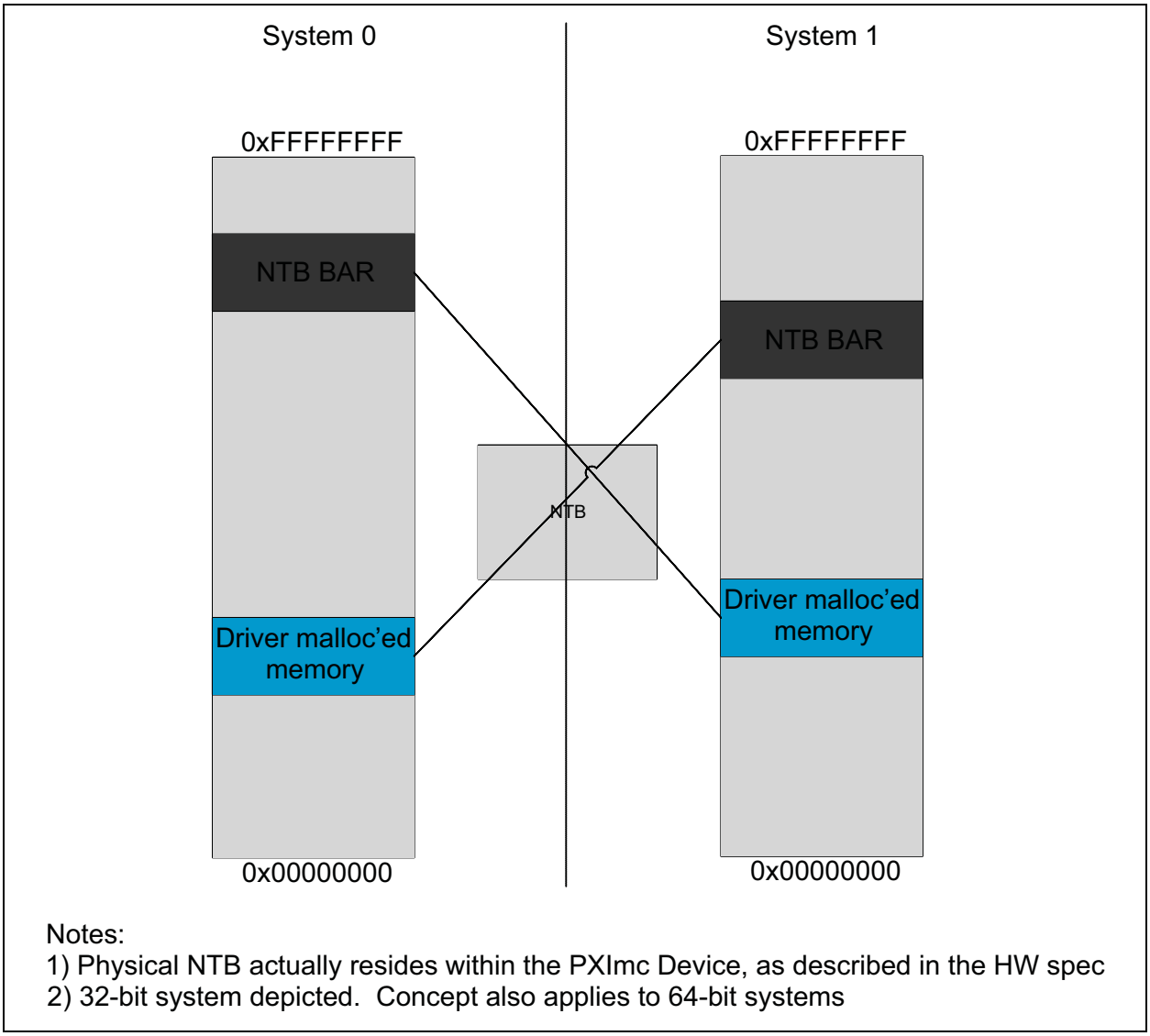


**Figure C-2.** Access Mapping of NTB BAR(s) to Physical Memory

# C.3 I/O via PXImc

The means for I/O over PXImc is that client processes are given temporary ownership of slices of address space in both the NTB BAR and/or the physical memory (RAM) that the NTB driver allocated. For example, if PXImc clients on both System 0 and System 1 want to communicate using some protocol (protocol 0x5000 for this example), the objective is to allow the client on System 0 to own a slice of the NTB BAR that gets mapped to physical memory on System 1, where its communication partner can also perform accesses to that physical memory. Also, the client on System 0 should be allowed to own a slice of the local physical memory (depicted in the figure by the "Driver malloc'ed memory") that can be targeted by System 1 accesses to a slice of System 1's BAR.

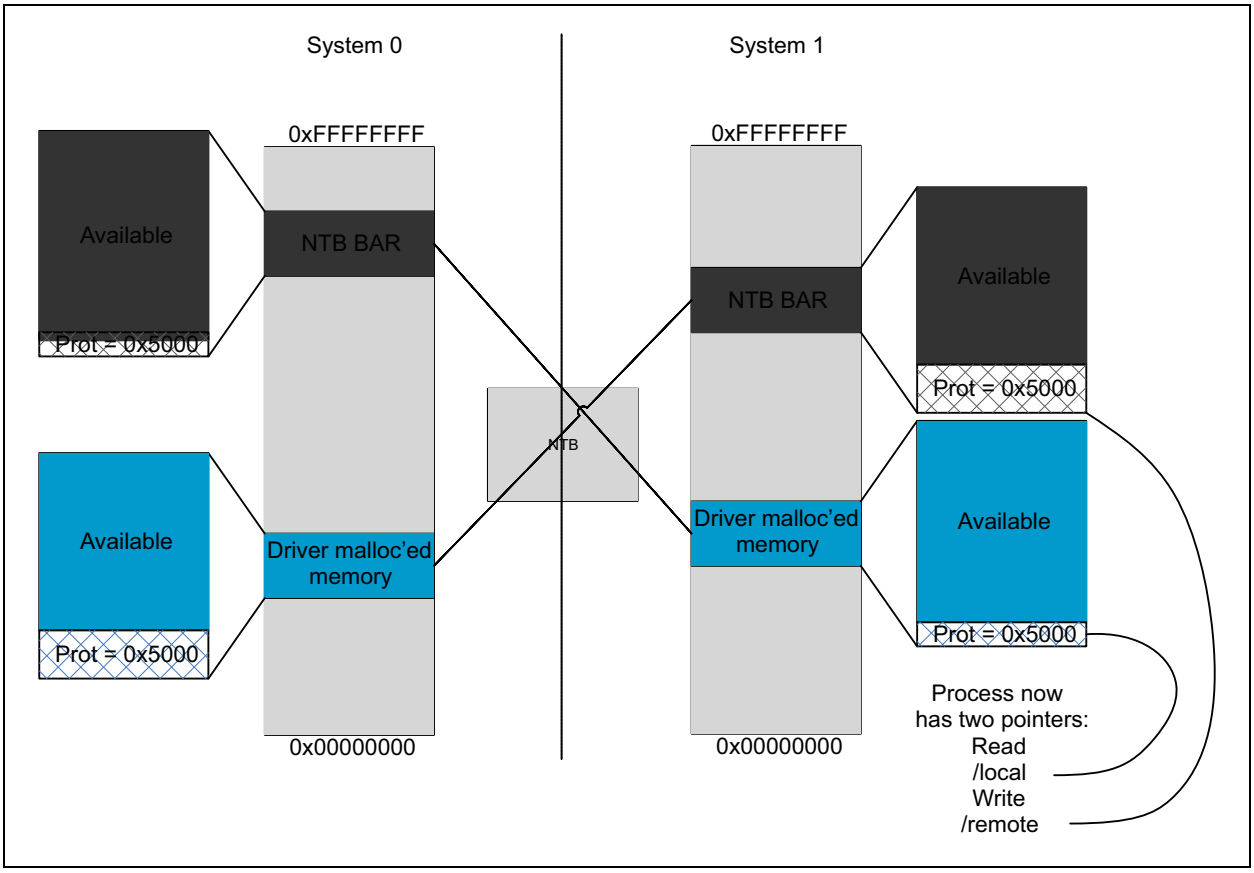


**Figure C-3.** I/O Via PXImc

The objective that communication initialization needs to achieve is matching a PXImc client on System 0 with a PXImc client on System 1, then determining how much physical memory on both systems the connection needs, and then granting ownership of that system memory, and of the associated BAR space, to the two paired PXImc clients. It's not a requirement that physical memory is needed on both systems, the two matched PXImc clients may decide to operate only within one system's physical memory.

The PXImc API allows a session to be paired with another session (through the session pairing criteria), and a way for this paired connection to be given memory on one or both sides. The PXImc Logic Block Vendor needs to abstract the details of dealing with BAR sizes, number of BARs, and how BARs get mapped to physical memory from the user of PXImc. These details are outside the scope of the specification; the Logic Block Vendor needs to handle them in some way such that the above functionality is possible.