VMEbus Extensions for Instrumentation

TCP/IP Instrument Protocol Specification

VXI-11

Revision 1.0

July 17, 1995

NOTICE

The information contained in this document is subject to change without notice.

The VXIbus Consortium, Inc. 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 VXIbus Consortium, Inc. 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.

VMEbus Extensions for Instrumentation TCP/IP Instrument Protocol Specification VXI-11, Revision 1.0 is authored by the VXIbus Consortium, Inc. and its sponsor members:

GenRad, Inc.
Hewlett-Packard Co.
National Instruments, Corp.
Racal Instruments, Inc.
Tektronix, Inc.
Wavetek, Inc.

Table of Contents

List of Tables

List of Figures

VMEbus Extensions for Instrumentation:

TCP/IP Instrument Protocol Specification

A. INTRODUCTION

The need to connect instruments to computer networks has developed in the test and measurement industry. The connections required may be to either local-area networks (LANs) or wide-area networks (WANs). Along with this comes the need to have a standard that specifies the interconnection of controllers and devices over a computer network. This specification, which is part of the VXIbus set of specifications, describes how instrumentation can be connected to industry-standard networks. The communications and programming paradigms supported by this specification are similar in nature to the techniques supported by IEEE 488.2. The protocol described allows ASCII-based communications to take place between a controller and a device over a computer network. The reader should be knowledgeable about networks, the Internet Protocol Suite, ONC RPC, and IEEE 488.2.

A.1. SCOPE

This specification is part of the VXIbus set of specifications and defines a *network instrument* protocol to be used for controller - device communication over a TCP/IP network.

The only networks directly considered by this specification are those which support the Internet Protocol Suite. The techniques defined in this specification could be used over other networks, such as networks which support the OSI protocol standards, but this document does not address that mapping. This specification uses Open Network Computing (ONC) remote procedure calls on top of the Internet Protocol Suite. The use of ONC/RPC is for the specification of the protocol on the network only, and does not specify a particular application interface.

Other network protocols may also be supported by a *network instrument* host.

A.2. DOCUMENT STRUCTURE

This document is divided into 2 sections. The first section, an introduction, is intended to familiarize readers with the intent and scope of the document.

The second section, *Network Instrument* Protocol, defines the network protocol to be used for communication between controllers and devices over a TCP/IP network.

A.3. SPECIFICATION OBJECTIVES

This specification has the following objectives:

1. To allow ASCII messages, including IEEE 488.2 messages, and IEEE 488.1 instrument control messages to be passed between a controller and a device over a TCP/IP network.
2. To define an instrument protocol which can be used for this controller/device communication over a TCP/IP network.
3. To enable the interconnection of independently manufactured apparatus into a single functional system.
4. To provide a mechanism to extend the protocol.
5. To define an instrument protocol which can support diverse application interfaces.
6. To allow for other networking protocols as the functionality of devices and controllers dictate, such as NFS or telnet.

A.4. DEFINITION of TERMS

controller: a component of the system which sends program messages to and receives response messages from one or more devices.

device: a uniquely addressable component of a system which receives program messages from and sends response messages to one or more controllers.

***network instrument* host:** an end-point on a network which may include controllers, devices, *network instrument* clients, or *network instrument* servers.

***network instrument* connection:** a connection between a *network instrument* client and a *network instrument* server which contains a core channel, optionally an abort channel, and optionally an interrupt channel.

***network instrument* client:** an entity which maintains a single *network instrument* connection with a *network instrument* server for one or more controllers.

***network instrument* server:** an entity which maintains a single *network instrument* connection with a *network instrument* client for one or more devices.

***network instrument* message:** a well defined sequence of bytes sent between a *network instrument* client and a *network instrument* server which contains a request or reply. The *network instrument* messages are defined using ONC/RPC.

link: an instance of a communication pathway over a network instrument connection between a controller and a device.

system: a group of devices and controllers interconnected by a network which supports the Internet Protocol Suite and the *network instrument* protocol defined herein.

The following terms are used to identify the contents of paragraphs, as in other VXIbus Specifications. These definitions are the same as those in IEEE 1155-1992.

RULE: Rules **SHALL** be followed to ensure compatibility. A rule is characterized by the use of the words **SHALL** and **SHALL NOT**. These words are not used for any other purpose other than stating rules.

RECOMMENDATION: Recommendations consist of advice to implementors which will affect the usability of the final device. Discussions of particular hardware to enhance throughput would fall under a recommendation. These should be followed to avoid problems and to obtain optimum performance.

PERMISSION: Permissions are included to 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. The word **MAY** is reserved for indicating permissions.

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.

Any text that appears without a heading should be considered as description of the standard.

A.5. REFERENCES

[1] IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation.
[2] IEEE Std 488.2-1992, IEEE Standard Codes, Formats, Protocols, and Common Commands For Use With IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation.
[3] Internet Protocol, Request for Comments 791, Jon B. Postel, DDN Network Information Center, SRI International, September, 1981, See also MIL-STD 1777.
[4] Transmission Control Protocol, Request for Comments 793, Jon B. Postel, DDN Network Information Center, SRI International, September, 1981, See also MIL-STD 1777.
[5] A Standard for the Transmission of IP Datagrams over Ethernet Networks, Request for Comments 894, C. Hornig, DDN Network Information Center, SRI International, April 1984.
[6] XDR: External Data Representation Standard, Request for Comments 1014, Sun Microsystems, DDN Network Information Center, SRI International, June, 1987.
[7] A Standard for the Transmission of IP Datagrams over IEEE 802 Networks, Request for Comments 1042, J. Postel and J. Reynolds, DDN Network Information Center, SRI International, February 1988.
[8] RPC: Remote Procedure Call Protocol Specification, Request for Comments 1057, Sun Microsystems, DDN Network Information Center, SRI International, June, 1988.
[9] Requirements for Internet Hosts -- Communication Layers, Request for Comments 1122, R. Braden, DDN Network Information Center, SRI International, October, 1989.
[10] ISO 8802-2:1989[ANSI/IEEE 802.2-1989] Information Technology - Local and Metropolitan Area Networks - Part 2: Logical Link Control.
[11] ISO/IEC 8802-3:1993 [ANSI/IEEE 802.3-1993] Information Technology - Local and Metropolitan Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.
[12] The Ethernet, Physical and Data Link Layer Specifications, Version 2.0, Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, 1982.

A.6. RELATED DOCUMENTS

This specification is one document in a set of specifications which describe a method for ASCII-based communication over a network between controllers and devices. This specification describes the protocol

used for this communication. Other specifications in the group describe the specific mapping from the protocol described here to the specific interface addressed by the companion specification. This specification does not require that the companion specifications be followed to claim compliance to this specification alone. This specification also does not require the existence of a companion specification.

The recommended approach for using these specifications is to first read this specification, since the protocol forms the foundation upon which the companion specifications are built, then the appropriate companion specification should be read. If your interest is in connecting VXIbus devices to a LAN, read the companion specification TCP/IP-VXIbus Interface Specification, VXI-11.1. If your interest is in connecting IEEE 488.1 devices to a LAN, read the companion specification TCP/IP-IEEE 488.1 Interface Specification, VXI-11.2. If your interest is in connecting IEEE 488.2 style instruments directly to a LAN, read the companion specification TCP/IP-IEEE 488.2 Instrument Interface Specification, VXI-11.3. If your interest is in connecting devices which support some other interface, such as RS-232, to a LAN, you may want to read one or more of the companion specifications in order to understand the general approach taken in mapping from the protocol to a specific interface.

Those specifications listed below are currently part of this group:

[1] VMEbus Extensions for Instrumentation: TCP/IP-VXIbus Interface Specification, VXI-11.1, Revision 1.0.
[2] VMEbus Extensions for Instrumentation: TCP/IP-IEEE 488.1 Interface Specification, VXI-11.2, Revision 1.0.
[3] VMEbus Extensions for Instrumentation: TCP/IP-IEEE 488.2 Instrument Interface Specification, VXI-11.3, Revision 1.0.

B. *NETWORK INSTRUMENT* PROTOCOL

The *network instrument* protocol uses the ONC remote procedure call (RPC) model. Conceptually, this model allows one application (typically called the client) to call procedures in a remote application (typically called the server) as if the remote procedures were local. This specification uses ONC/RPC for defining the *network instrument* messages which are passed over the network, but does not require that these RPCs be provided as the application interface. The ONC/RPC interface may, however, be used by a device designer as a matter of convenience.

The client identifies the remote procedure, or message, by a unique number. This number is then encoded into a message along with the procedure's argument types and values. The message is sent to the server machine where it is decoded by the server. The server uses the unique identifier to dispatch the request. When the request is completed, the return values are encoded into a message which is sent back to the client machine.

The interface definition (see Appendix I, "*Network instrument* RPCL") gives the function prototypes as well as the unique identifiers for the procedures. For ONC RPC, the unique identifier is a combination of a program number (also known as an interface id), a procedure number, and a version number.

Table B.1 outlines the 17 messages that define the *network instrument* protocol. These messages are expected to be supported by all devices that claim to be *network instrument* compliant. Most of these messages will be familiar to those who have worked with IEEE 488 devices.

Message	Channel	Description
create_link	core	opens a link to a device
device_write	core	device receives a message
device_read	core	device returns a result
device_readstb	core	device returns its status byte
device_trigger	core	device executes a trigger
device_clear	core	device clears itself
device_remote	core	device disables its front panel
device_local	core	device enables its front panel
device_lock	core	device is locked
device_unlock	core	device is unlocked
create_intr_chan	core	device creates interrupt channel
destroy_intr_chan	core	device destroys interrupt channel
device_enable_srq	core	device enables/disables sending of service requests
device_docmd	core	device executes a command
destroy_link	core	closes a link to a device
device_abort	abort	device aborts an in-progress call
device_intr_srq	interrupt	used by device to send a service request

Table B.1 *Network instrument* Protocol

The messages are sent over three different channels: a core synchronous command channel, a secondary abort channel (for aborting core channel operations), and an interrupt channel.

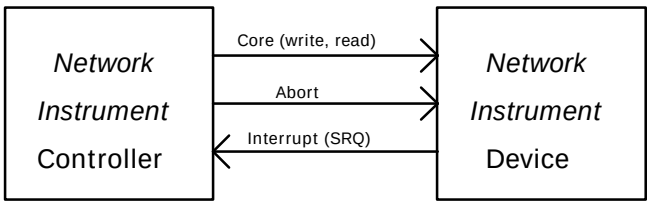


Figure B.1 *Network instrument* Channels

B.1. PROTOCOL FOUNDATIONS

The terminology used in this section to describe the stack used by the *network instrument* protocol will loosely follow the International Standards Organization (ISO) Open Systems Interconnection (OSI) reference model. The OSI model is a seven layer model depicted in the following figure.

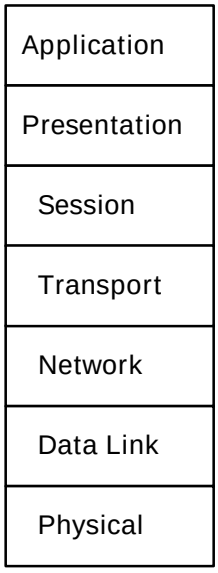


Figure B.2 OSI Reference Model

The *network instrument* protocol is an application layer protocol designed for controller to device communication using a paradigm similar to communication with IEEE 488 devices. The *network instrument* protocol is built on an industry-standard protocol stack, allowing instrumentation and controllers to communicate over existing networks. The following sections work from the bottom of the protocol stack upward, describing the protocol stack requirements of controllers or devices which implement the *network instrument* protocol.

B.1.1. Physical and Data Link Layer Requirements

RULE B.1.1:

Hosts **SHALL** support an Ethernet/802.3 Data Link Layer and an 802.3/10BASE-T Physical Layer. The device **SHALL** include an RJ-45 connector for 10BASE-T.

PERMISSION B.1.1:

Hosts **MAY** support other Data Link and Physical Layers in addition to Ethernet/802.3/10BASE-T.

B.1.2. Network and Transport Layer Requirements

RULE B.1.2:
Hosts **SHALL** support the Internet Protocol Suite, including the Transmission Control Protocol (TCP) and Internet Protocol (IP), and TCP **SHALL** be used as the transport layer.

PERMISSION B.1.2:
Hosts **MAY** also support UDP on the interrupt channel. See the description of the *create_intr_chan* message for more information.

OBSERVATION B.1.1:
The *network instrument* protocol is designed to use a reliable, connection-oriented transport service. Instrument procedures are generally not idempotent[1], as receiving the same message a second time may put the instrument into a different state, and the use of TCP ensures that operations are performed at most once. Using TCP implies that both sides of the connection are:

1. aware of the connection
2. and can detect when the connection terminates

The use of TCP as the underlying transport mechanism also:

1. ensures that messages are delivered in order
2. ensures that if a response is received, the procedure was executed exactly once
3. removes from the controller and device the need to verify the arrival of messages
4. allows parameters to a remote procedure to be any size

RFC 1122 of the Internet Engineering Task Force (IETF) outlines the requirements of hosts which support the Internet Protocol Suite.

The mechanism by which the TCP/IP stack is configured, including the IP address, is implementation dependent. The IP address and other stack parameters may be assigned using a suitable network protocol or configured using a local mechanism.

B.1.3. Session and Presentation Layer Requirements

RULE B.1.3:
All *network instrument* hosts **SHALL** implement a protocol whose messages are compatible with the Open Network Computing (ONC) remote procedure call (RPC) definition. This includes the use of the RPC mechanism at the session layer, and the use of the external data representation (XDR) mechanism at the presentation layer.

OBSERVATION B.1.2:
Network instrument hosts are not required to support ONC/RPC as an application interface. *Network instrument* hosts need only ensure that the messages sent and received as part of the *network instrument* protocol are ONC/RPC and XDR compatible.

---

[1] An idempotent procedure can be executed more than once without altering the device's state or its reply to the controller.

RULE B.1.4:
All hosts acting as *network instrument* servers **SHALL** support a port mapper.

A port mapper provides a means for a *network instrument* client to determine which port a *network instrument* server is listening on.

B.1.4. Application Layer Requirements

RULE B.1.5:
Network instrument hosts **SHALL** implement all the *network instrument* messages and their data types as defined in section B.5, "Basic Data Types", and section B.6, "Network Instrument Messages (RPCs)".

A *network instrument* message is a well defined sequence of bytes sent between a *network instrument* client and a *network instrument* server which contains a request or reply. The *network instrument* messages are defined using ONC/RPC.

B.1.5. Protocol Stack Summary

Based on the requirements at each layer, the resulting protocol stack appears in Figure B.3.

Application	*Network Instrument* as specified in this spec	
Presentation	XDR	RFC 1014
Session	ONC/RPC	RFC 1057
Transport	TCP	RFC 793
Network	IP	RFC 791
Data Link	Ethernet/802.3	8802-3
Physical	802.3/10BASE-T	8802-3

Figure B.3 *Network instrument* Protocol Stack

B.1.6. Network Instrument Client

A *network instrument* client is any entity which has a single *network instrument* connection to a *network instrument* server. A *network instrument* client may be a host, a process running on a host, or a thread running within a process on a host. This could affect the number of *network instrument* clients residing on any given host, and therefore the number of connections from that host.

B.1.7. Network Instrument Server

A *network instrument* server is any entity which has a single *network instrument* connection to a *network instrument* client. A *network instrument* server may be a host, a process running on a host, or a thread running within a process on a host. This could affect the number of *network instrument* servers residing on any given host, and therefore the number of available *network instrument* connections.

B.2. CONNECTION MODEL

This section defines the connection model of the *network instrument* protocol, as well as the relationship between controllers, devices, *network instrument* clients, and *network instrument* servers. The term *controller*, as used in this specification, typically refers to the RPC client, while the term *device* typically refers to the RPC server. The only exception is when the roles are reversed for interrupts, which will be described further later in the specification.

As discussed in the overview at the beginning of this specification and shown in Figure B.1, the *network instrument* protocol uses up to three channels between the controller and the device for passing *network instrument* messages. A *network instrument* connection is this set of channels:

- **Core Channel**: Used to transfer all requests except the *device_abort* RPC and the *device_intr_srq* RPC.
- **Abort Channel**: Used to transfer the *device_abort* RPC (optional for client).
- **Interrupt Channel**: Used to transfer the *device_intr_srq* RPC from the device to the controller (optional for client).

These three channels correspond to three RPC clients/servers.

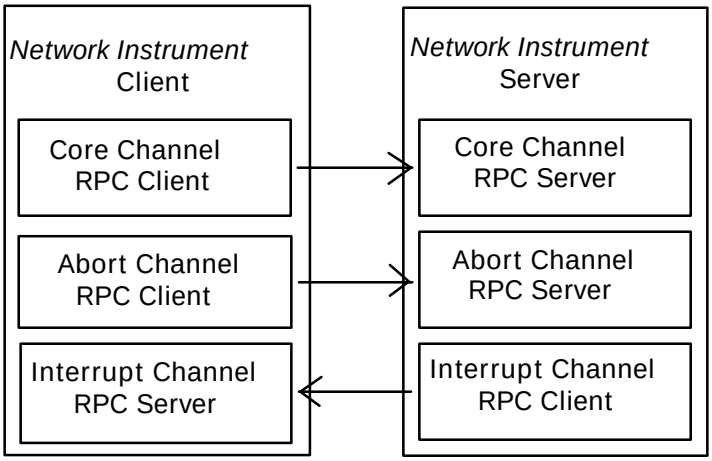


Figure B.4 *Network instrument* Channels

Network instrument connections are established using the create RPC client and create RPC server operations discussed in sections B.2.4 and B.2.5. With all three channels established a *network instrument* client contains two RPC clients and one RPC server, while a *network instrument* server contains two RPC servers and one RPC client.

RULE B.2.1:

A *network instrument* server **SHALL** implement all three of the channels described, and the channels **SHALL** be established by the defined connection establishment sequences outlined in sections B.2.4, Core and Abort Connection Establishment Sequence, and B.2.5, Interrupt Connection Establishment Sequence.

OBSERVATION B.2.1:

Although a *network instrument* server is required to support all three channels, a particular *network instrument* connection may not contain all three at the discretion of the *network instrument* client.

Links represent an instance of a communication pathway between a controller and a device. Any given *network instrument* connection may carry multiple links created with the *create_link* RPC. Also note that more than one controller may have a link open to a single device at the same time.

Figure B.5 shows a typical scenario with one *network instrument* client talking to one *network instrument* server. A single link is being used on a single connection to communicate with one device.

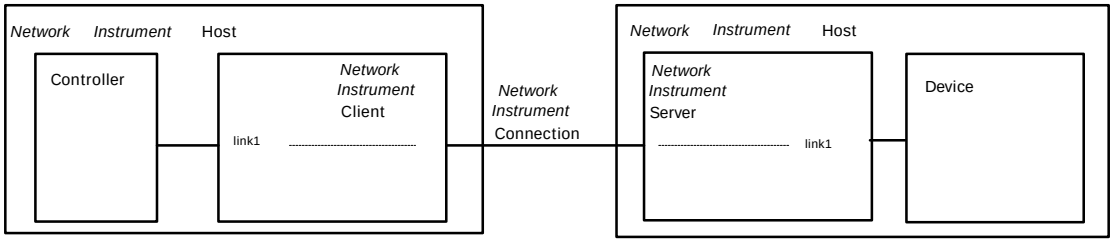


Figure B.5 Connection Model - Single Connection, One Device

Figure B.6 shows a typical scenario with one *network instrument* client talking to one *network instrument* server. Multiple links are being used on the single connection to communicate with more than one device at a time.

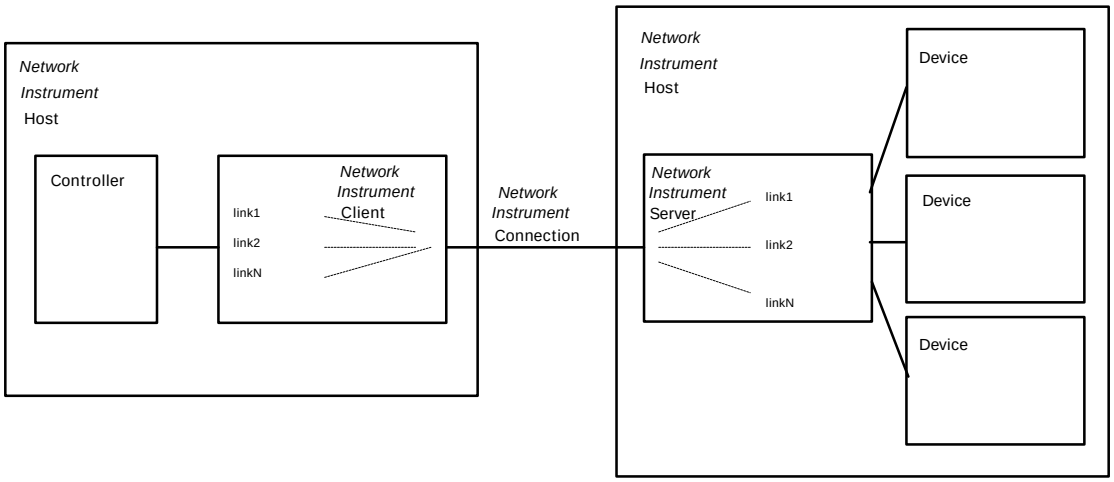


Figure B.6 Connection Model - Single Connection, Multiple Devices

Figure B.7 shows a scenario where a single host has two *network instrument* clients speaking with two *network instrument* servers.

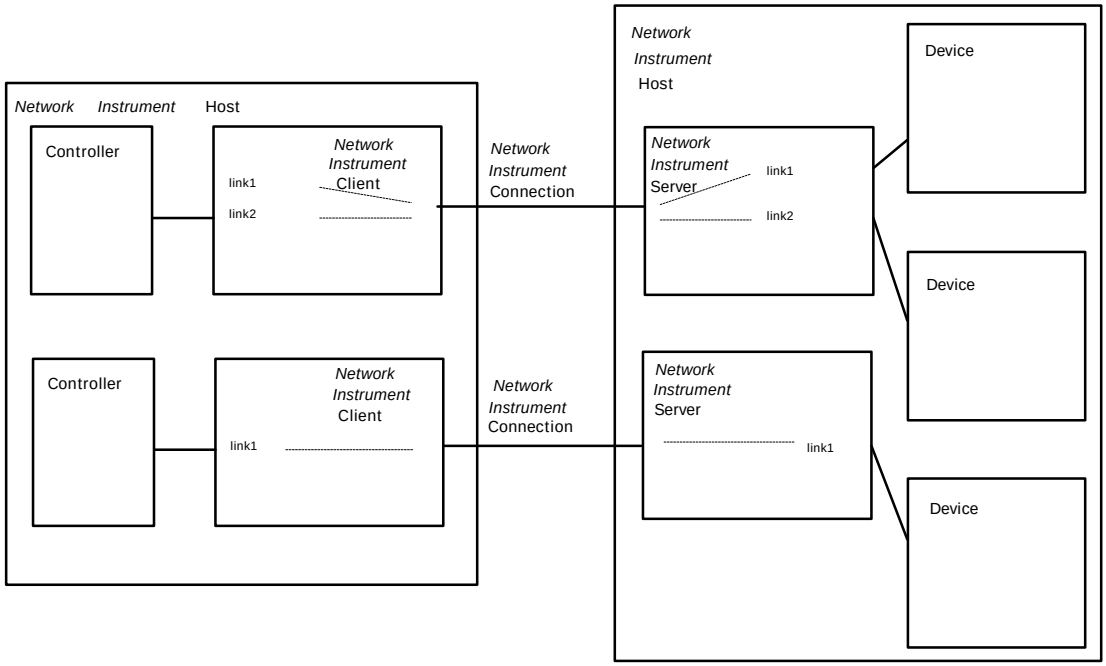


Figure B.7 Connection Model - Two Connections

Figure B.8 shows a scenario where two hosts both have links to the same device.

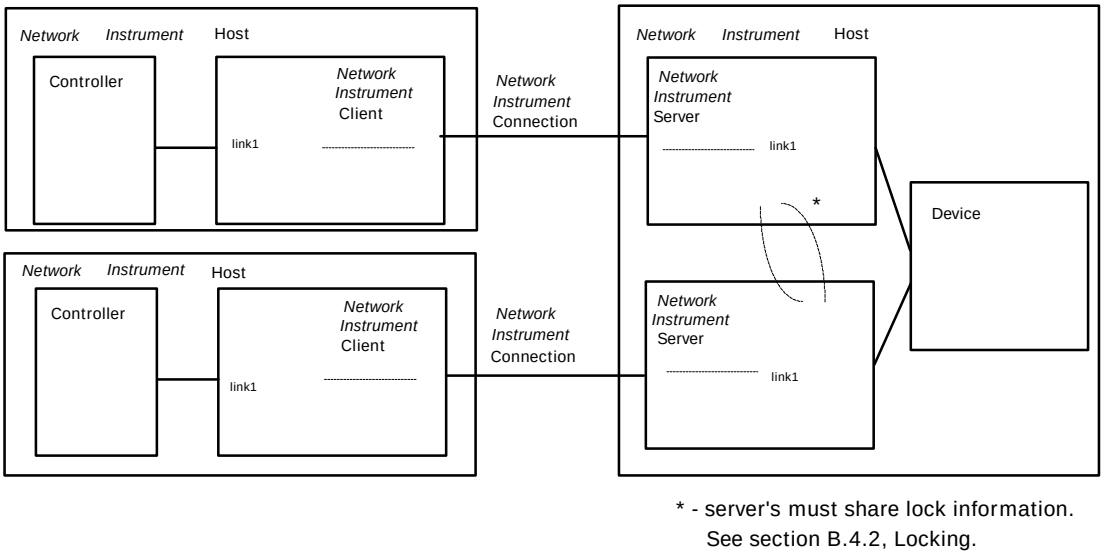


Figure B.8 Connection Model - Two Hosts, Single Device

Figure B.9 shows a scenario where two hosts are communicating with different devices in the same *network instrument* host.

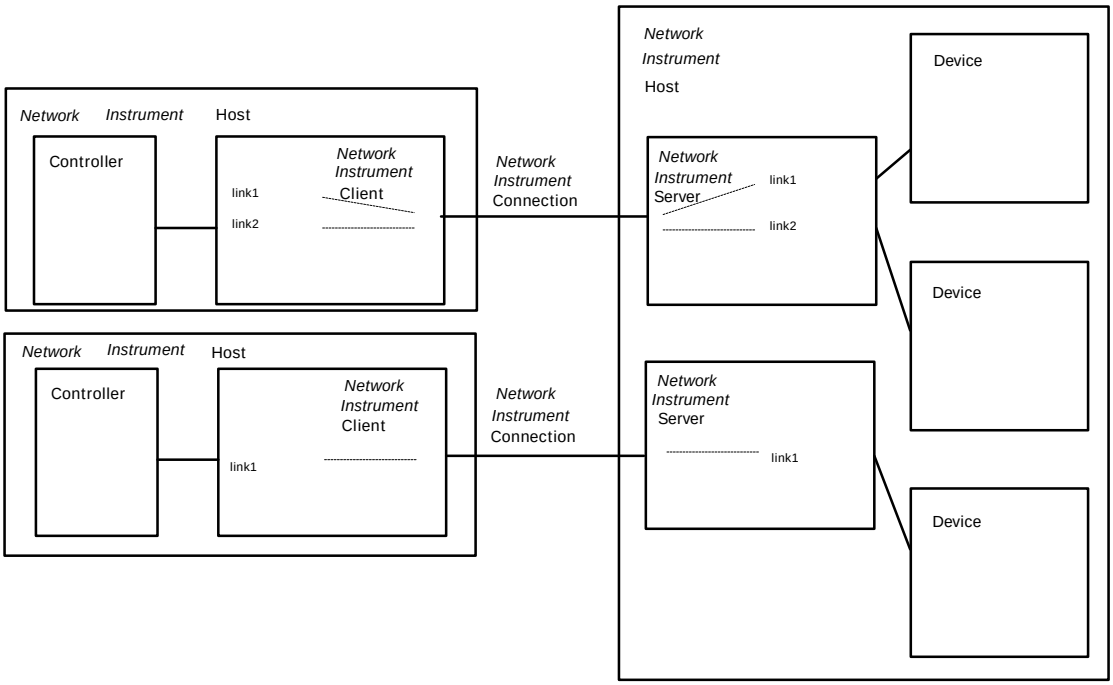
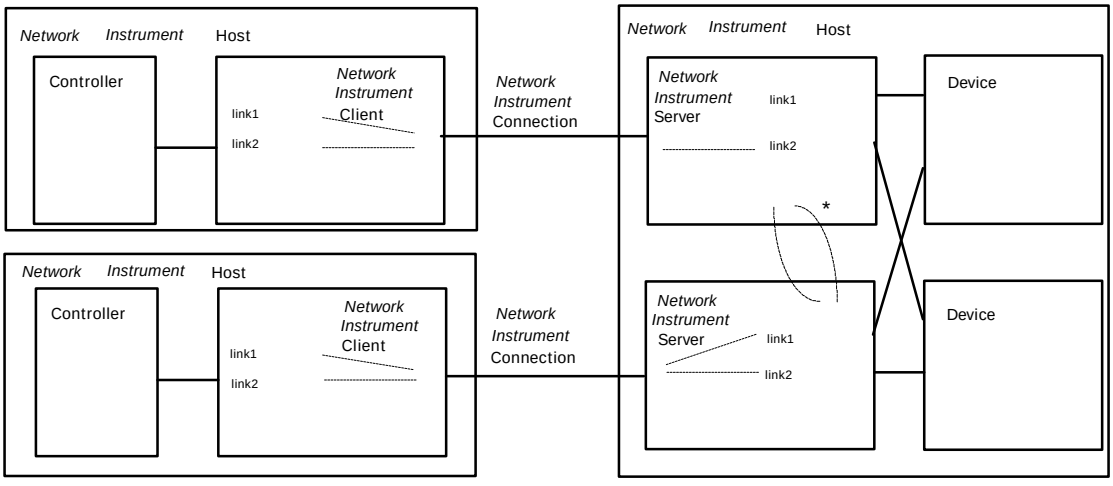


Figure B.9 Connection Model - Two Hosts, Multiple Devices

Figure B.10 shows a scenario where two hosts both have links to multiple devices within the same *network instrument* host



* - server's must share lock information.
See section B.4.2, Locking.

Figure B.10 Connection Model - Two Hosts, Concurrent Multiple Devices

Figure B.11 shows an invalid connection model. Two *network instrument* clients cannot be connected to the same *network instrument* server. This is due to the one-to-one nature of the TCP connections which are used between *network instrument* clients and servers.

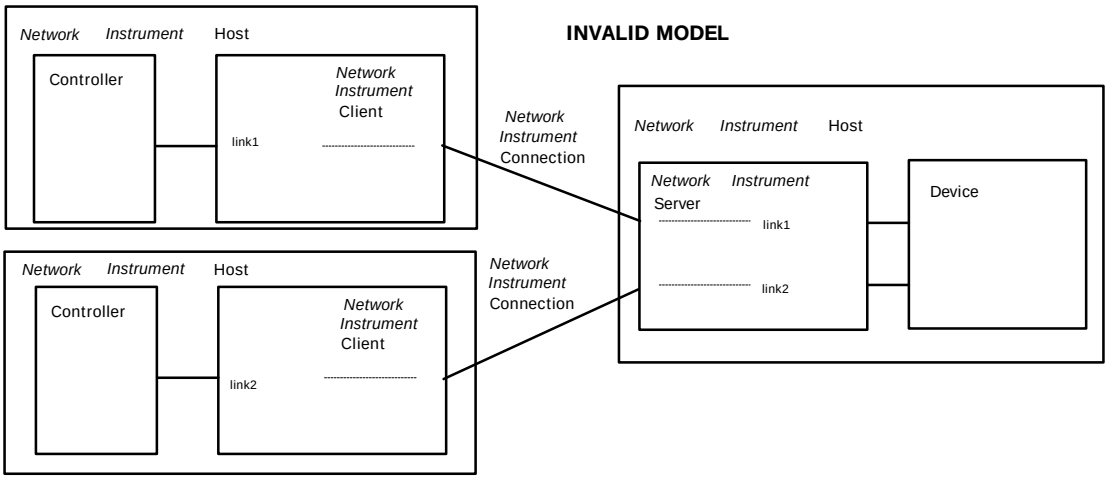


Figure B.11 Invalid Connection Model - Two Hosts

B.2.1. Core Channel

The core channel is used for all RPCs except abort and interrupts.

RULE B.2.2:

The *network instrument* server **SHALL** process all RPCs on this channel in the order received. The RPC reply **SHALL NOT** be sent until the associated action is complete.

OBSERVATION B.2.2:

The use of TCP on this channel ensures that messages arrive in order from the *network instrument* client and that messages sent from the *network instrument* server arrive at the *network instrument* client in order.

RULE B.2.3:

Other protocols carried on the core channel using different program numbers **SHALL NOT** interfere with the *network instrument* protocol.

OBSERVATION B.2.3:

This rule is not easily tested, but implementors should be aware that additional protocols which interfere with the *network instrument* protocol will cause system problems which are difficult to resolve.

RULE B.2.4:

All RPCs executed by a *network instrument* server which perform I/O to the same device or interface **SHALL** be serialized by the *network instrument* server.

OBSERVATION B.2.4:

RPCs which are executed entirely within the *network instrument* server are unaffected by this rule. RPCs acting on different devices and interfaces need not be serialized.

B.2.2. Abort Channel

RULE B.2.5:
The abort channel **SHALL** be used to carry only the *device_abort* RPC.

RECOMMENDATION B.2.1:
The *device_abort* RPC on the abort channel should be responded to in a timely manner.

OBSERVATION B.2.5:
A *network instrument* server's abort channel is typically implemented as an interrupt or signal handler in a single threaded operating system, or as a higher priority thread in a multi-threaded operating system.

B.2.3. Interrupt Channel

The interrupt channel is used by the *network instrument* server to deliver service requests to the *network instrument* client. This effectively reverses the role of client and server. The *network instrument* server acts as an RPC client, making a remote procedure request of the *network instrument* client, acting as an RPC server.

RULE B.2.6:
The interrupt channel **SHALL** be established by the *network instrument* server to the *network instrument* client after the *network instrument* client issues the *create_intr_chan* RPC.

OBSERVATION B.2.6:
A *network instrument* client's interrupt channel is typically implemented as an interrupt or signal handler in a single threaded operating system, or as a separate thread in a multi-threaded operating system.

B.2.4. Core and Abort Channel Establishment Sequence

Figure B.12 describes the order in which the connection establishment typically takes place for the core and abort channels. Note that the second and third *create_link* request/reply pairs are listed in the figure only to emphasize that the same port number is returned on subsequent *create_link*s after the first, and that no additional channel creation is necessary after the first *create_link* sequence is complete.

***Network Instrument* Client**	***Network Instrument* Server**
	create RPC server (abort channel) create RPC server (core channel) register core channel with portmapper be ready to accept connection requests
create RPC client (core channel.) *create_link*(1)	
	reply to *create_link*(1) - return abort port #
create RPC client (abort chan., optional)	
create_link(2)	
	reply to *create_link*(2) - return same abort port #
create_link(3)	
	reply to *create_link*(3) - return same abort port #

Figure B.12 Core and Abort Channel Establishment Sequence

OBSERVATION B.2.7:

The steps mentioned in Figure B.12 involve the following. Implementation details may vary from one operating system to another.

- create RPC server (abort/core)- create listen socket upon which connection requests will be accepted and set up any local data structures required to track the RPC server, typically done by a *svctcp_create*.
- register core channel with port mapper - register the program number and version number with the local port mapper, typically associated with the *svc_register*, which also sets up local data structures to dispatch requests.
- create RPC client (core/abort)- temporarily connect to the port mapper on the server to find the port for the program number and version being used by the *network instrument* protocol. After determining the port number, create the core channel by connecting to that port. Set up any local data structures necessary to track the RPC client. This step is typically done by a *clnttcp_create*.
- *create_link* requests and replies - These steps represent sending *network instrument* protocol *create_link* requests and replies.

After the first *create_link*, the *network instrument* client may create an RPC client for the abort channel, but no additional client creations are necessary after subsequent *create_links*. These connections may be torn down by the *network instrument* client once all links have been closed with *destroy_link*. The whole sequence could then start over.

RULE B.2.7:

The *network instrument* server **SHALL** return the same abort port number in all replies to *create_link* sent on the same core channel.

If the *network instrument* client establishes an abort channel, the port number returned in the *create_link* reply is used to make the connection.

OBSERVATION B.2.8:

The *network instrument* client is not required to create an abort channel.

RULE B.2.8:

The *network instrument* server **SHALL** accept and process RPCs on the core channel even if an abort channel is never established.

B.2.5. Interrupt Channel Establishment Sequence

Figure B.13 describes the process by which the interrupt channel is established.

***Network Instrument* Client**	***Network Instrument* Server**
create RPC server (interrupt channel) *create_intr_chan*	
	create RPC client (interrupt channel) reply to *create_intr_chan*

Figure B.13 Interrupt Channel Establishment Sequence

RULE B.2.9:

A *create_intr_chan* request received when an interrupt channel already exists **SHALL NOT** cause the *network instrument* server to create a new channel. The *network instrument* server **SHALL** use the existing interrupt channel such that there is only one interrupt channel used by all links on that *network instrument* connection.

RULE B.2.10:

If the *network instrument* client issues *destroy_intr_chan*, then the *network instrument* server **SHALL** destroy the RPC client to tear down the interrupt channel.

OBSERVATION B.2.9:

If the *network instrument* client never calls *destroy_intr_chan*, the interrupt channel is closed by the *network instrument* server when the core channel is closed by the *network instrument* client.

B.3. INTERRUPT LOGIC

The interrupt mechanism allows the device to send a notification call to the controller (effectively switching the roles of RPC client and RPC server). One way a controller could implement the interrupt mechanism is to register a handler for the interrupt, inform the current device to enable the interrupt, and then service the interrupt when it occurs. Figures B.14 and B.15 shows possible sequences of interrupt channel creation and usage.

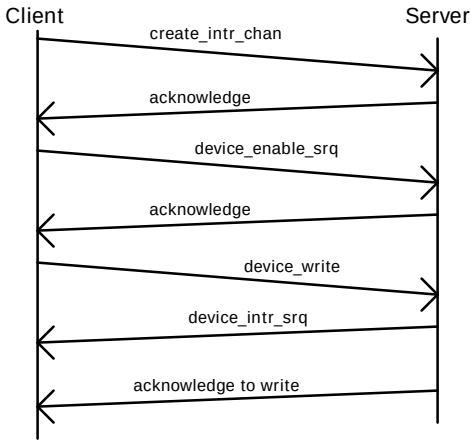


Figure B.14 Interrupts - SRQ in the middle of another call

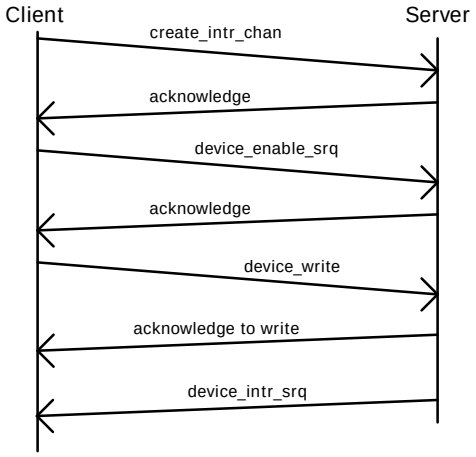


Figure B.15 Interrupts - SRQ after another call

Network instrument clients can implement interrupts by using either a separate interrupt process, threads, or by emulating threads using a signal handling routine that is invoked on incoming messages to the interrupt port.

PERMISSION B.3.1:

The *network instrument* server **MAY** issue interrupts in the middle of an active call. In general, this implementation gives more timely responses, and can be easier than delaying the interrupt until an in-progress action has finished.

The *device_intr_srq* RPC is implemented as a one-way RPC. This means that the *network instrument* server does not expect a response from the *network instrument* client. This is necessary to avoid deadlock situations in a single-threaded environment where if a response were expected to an interrupt both the *network instrument* client and *network instrument* server could be waiting for a response from the other, with neither proceeding.

The *create_intr_chan* RPC is used to identify the host or port that can service the interrupt. The *device_enable_srq* RPC is used to enable or disable an interrupt. The destroy_intr_chan RPC is used to close the interrupt channel.

OBSERVATION B.3.1:

The *device_enable_srq* RPC contains a *handle* parameter. The same data contained in *handle* is passed back in the *handle* parameter of the *device_intr_srq* RPC. Since the same data is passed back, the *network instrument* client can identify the link associated with the *device_intr_srq*.

The *network instrument* protocol recognizes one type of interrupt, service request. Note that the return type to the interrupt RPC is void, denoting a one-way RPC.

RULE B.3.1:

A *network instrument* host **SHALL** use the following RPCL definition for interrupt messages.

```
struct Device_SrqParms {
    opaque  handle<>;
};

program DEVICE_INTR {
    version DEVICE_INTR_VERSION {
        void device_intr_srq (Device_SrqParms)   = 30;
     }=1;
}= 0x0607B1;
```

The program number 0x0607B1 is the registered program number for the *network instrument* protocol's interrupt channel.

B.4. SYSTEM BEHAVIOR

B.4.1. Multiple Controllers

A typical communication model consists of a single path from a controller to a device. On an RPC interface multiple controllers may be connected to the device’s RPC interface at any given time. Therefore multiple controllers may be attempting to write to the device's input buffer and read from the device's output buffer at the same time. The device may also be sending service requests to more than one controller. The controllers accessing the device should take actions, such as locking, to prevent corruption of each others transactions, or unpredictable results may occur.

OBSERVATION: B.4.1:

Instruments may not readily support being controlled by more than one controller due to the complexities of identifying atomic actions and interactions among instrument commands.

RECOMMENDATION B.4.1:

An instrument's host should support at least two *network instrument* servers simultaneously.

OBSERVATION B.4.2:

The previous recommendation does not imply that the instrument's host will have at least two processes waiting to receive connection requests. A common implementation involves dynamic creation of the *network instrument* servers, where a *network instrument* server is created when a connection request arrives.

RECOMMENDATION B.4.2:
An instrument's host should support simultaneous routing of at least two links to any device. The links may be through the same or different *network instrument* servers.

RULE B.4.1:
A *network instrument* server **SHALL** support links to every device accessible to any other *network instrument* server in the host.

OBSERVATION B.4.3:
The intent of this rule is to prevent the dedication of particular devices to particular *network instrument* servers.

B.4.2. Locking

In topologies as seen in figure B.10 a single device may be accessed by multiple controllers over separate links. For these situations the network instrument server supports locking access to a link, which guarantees exclusive access to the device associated with that link to that link only.

If a controller expects to have exclusive access to a device, it must have the lock. When no link has the lock, multiple controllers may be sending data and generally manipulating the state of the device. Under such circumstances, the behavior of the device is unpredictable.

The first call to *device_lock* for an unlocked device acquires the lock. Subsequent calls to *device_lock* for the same device return an error. *device_unlock* unlocks the device if this link has the lock, otherwise *device_unlock* returns an error.

OBSERVATION B.4.4:
Care should be taken in implementing locking to ensure that multiple *network instrument* servers do not believe they have acquired a lock simultaneously.

B.4.3. Time-outs

Many of the remote procedures are passed timeout values. Values may be specified for I/O operations and locks

RULE B.4.2:
The *network instrument* server **SHALL** allow at least *io_timeout* milliseconds for an I/O operation to complete.

OBSERVATION B.4.5:
The time it takes for the I/O operation to complete does not include any time spent waiting for the lock.

PERMISSION B.4.1:
An *io_timeout* of zero **MAY** be interpreted by the *network instrument* server to mean that the associated I/O operation should not block.

RULE B.4.3:
If the device is locked by another link and the *lock_timeout* is non-zero, the *network instrument* server **SHALL** allow at least *lock_timeout* milliseconds for a lock to be released. If the device is locked by another link and the *lock_timeout* is equal to zero, the *network instrument* server **SHALL NOT** wait for a lock to be released, but **SHALL** return an error immediately.

PERMISSION B.4.2:
No requirements are made on the precision of the time. A *network instrument* server **MAY** round any *io_timeout* value and any non-zero *lock_timeout* value up to a value consistent with the timing precision within the *network instrument* server.

RULE B.4.4:
A *network instrument* client **SHALL** provide a client side (local) timeout mechanism which is used in the event that the *network instrument* server does not respond in the specified amount of time.

This timeout mechanism is typically provided by the RPC subsystem. How this timeout value is set and what values it may take depend on aspects of the RPC subsystem beyond the scope of this specification.

OBSERVATION B.4.6:
The RPC client side (local) timeout value should be set to a value greater than the sum of the *io_timeout* and *lock_timeout* values passed to the *network instrument* server. If the RPC client timeout is too short, the client may timeout and stop listening for a reply prior to the *network instrument* server successfully completing the requested operation and replying.

OBSERVATION B.4.7:
A reply sent by the *network instrument* server after the *network instrument* client is no longer listening for it should be discarded by the *network instrument* client when it next sends or receives a *network instrument* message (and this is typically handled by the RPC subsystem, if such a subsystem is used by a particular implementation).

B.4.4. Dropped or Broken Connections

RULE B.4.5:
When the core channel is reset or closed (as defined by TCP), the *network instrument* server **SHALL** recognize this condition and release all resources associated with all links which were active on that *network instrument* connection (as if a *destroy_link* was executed for each open link on that connection). Resources to be released include locks, the abort channel, and the interrupt channel.

RECOMMENDATION B.4.3:
The *network instrument* server should also be configured to use an implementation defined mechanism to discover if the network is down or if a *network instrument* client has crashed, and perform the same cleanup actions.

B.4.5. Security Control

The RPC interface defined by this specification provides no services to authenticate a user for security. A controller must merely know a *network instrument* host's IP address to access all of its functions. Security control is beyond the scope of this specification, though a *network instrument* host may support security control methods.

B.4.6. Concurrent Operations

The protocol defined by this specification does not preclude the *network instrument* server or *network instrument* client from attempting to perform operations concurrently. However, due to the nature of most commercially available RPC software packages which may be used to implement the protocol defined by

this specification, it is expected that a typical *network instrument* host's implementation will serialize the RPCs.

If a *network instrument* client's implementation does allow multiple RPCs to be outstanding, then the *network instrument* server on the receiving end may have multiple RPC requests queued in its TCP input buffer. These RPCs are pending, but not in progress, and therefore are unaffected by *device_abort*.

B.5. BASIC DATA TYPES

The following XDR definitions are used by many of the RPC definitions that follow.

B.5.1. Device_Link

The *network instrument server* returns an identifier of type *Device_Link* as a result of the *create_link* call. This identifier is handed back to the *network instrument* server by the *network instrument* client on each subsequent call.

```
typedef long Device_Link;
```

The *network instrument server* verifies the validity of the identifier on each call.

The *Device_Link* data is not modified by the controller.

B.5.2. Error Codes

The result of any remote procedure call is a data structure whose first element has the type of *Device_ErrorCode*. A value of *0* indicates that the call was successfully completed and the results are valid. Any other value indicates that during the execution of the call, the *network instrument* server detected an error. All other error codes are reserved.

```
typedef long Device_ErrorCode ;
struct Device_Error {
    Device_ErrorCode error;
};
```

Table B.2 lists the possible error codes.

error	Meaning
0	No error
1	Syntax error
3	device not accessible
4	invalid link identifier
5	parameter error
6	channel not established
8	operation not supported
9	out of resources
11	device locked by another link
12	no lock held by this link
15	I/O timeout
17	I/O error
21	Invalid address
23	abort
29	channel already established

Table B.2 *error* Values

B.5.3. Operation Flags

The operation flags are passed on many of the calls to communicate additional information concerning how the request is carried out. Undefined bits are reserved for future use. Controllers send undefined bits as zero (0). These flags are sent from the *network instrument* client to the *network instrument* server as parameters to several of the RPCs.

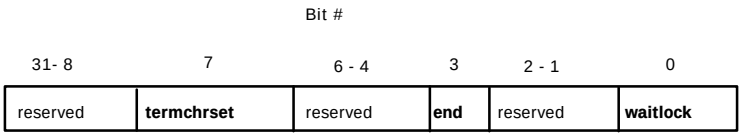


Figure B.16 Operation Flags

- **waitlock**(1_{16}): If the flag is set to one(1), then the *network instrument* server suspends (blocks) the requested operation if it cannot be performed due to a lock held by another link for at least *lock_timeout* milliseconds. If the flag is reset to zero(0), then the *network instrument* server sets the *error* value to 11 and returns if the operation cannot be performed due to a lock held by another link.
- **end**(8_{16}): If the flag is set to one(1) then the last byte in the buffer is sent with an END indicator. This flag is only valid for *device_write*.
- **termchrset**(80_{16}): This flag is set to one(1) if a termination character is specified on a read. The actual termination character itself is passed in the *termchr* parameter. This flag is only valid for *device_read*.

```
typedef long Device_Flags;
```

B.5.4. Timeouts

The *io_timeout* value determines how long a *network instrument* server allows an I/O operation to take. The *lock_timeout* determines how long a *network instrument* server will wait for a lock to be released. Units for both are in milliseconds.

```
   unsigned long io_timeout;   /* time to wait for I/O */
   unsigned long lock_timeout; /* time to wait for a lock */
```

B.5.5. Generic Parameter

The generic parameter is used by several of the RPCs to pass the link ID, the operation flags, and the timeout value to the device.

```
struct Device_GenericParms {
   Device_Link      lid;     /* Device_Link ID from create_link */
   Device_Flags     flags;   /* flags with options */
   unsigned long    lock_timeout; /* time to wait for lock */
   unsigned long    io_timeout;   /* time to wait for I/O */
};
```

B.5.6. XDR ints and longs

The XDR encoding and decoding allows for integers to be passed between hosts, even when those hosts have different integer representations. All integers defined by this specification are passed over the network as 32-bit integers, either signed or unsigned as defined.

B.5.7. Opaque Arrays

In the case of *device_write* and *device_read* the XDR opaque type is used by the network instrument protocol not because the data being represented is truly opaque, but to avoid the overhead associated with character data (8 bits being promoted to 32 bits). Since the *data* parameters for *device_write* and *device_read* are arrays, a structure is passed which contains a pointer to the data, *data.data_val*, and the number of elements, *data.data_len*.

B.6. Network Instrument Messages (RPCs)

The following sections describe the actions each remote procedure performs. For each procedure:

1. The required functionality of the procedure is described. Successful completion of the procedure, indicated by setting *error* to zero (0), means the required functions were performed.
2. Error conditions are described. Which error number to return under various conditions is given.
3. The procedure definition is given using RPCL (Remote Procedure Call Language).

OBSERVATION B.6.1:

The RPCL definitions in this section should match the ones in section C, but the definitions in section C take precedence.

RULE B.6.1:

The program and version numbers shown in Table B.3 **SHALL** be used by *network instrument* servers and *network instrument* clients for the *network instrument* protocol.

	Program Number	Version	Protocol
Core Channel	395183	1	TCP
Abort Channel	395184	1	TCP
Interrupt Channel	395185	1	TCP

Table B.3 Program Numbers

Section C contains the complete RPCL description of the *network instrument* protocol.

RULE B.6.2:

Only the defined procedure numbers **SHALL** be used with the program numbers listed in Table B.3.

All other procedure numbers are reserved for future use.

B.6.1. create_link

The *create_link* RPC creates a new link. This link is identified on subsequent RPCs by the *lid* returned from the *network instrument* server.

```
struct Create_LinkParms {
   long          clientId;     /* implementation specific value.*/
   bool          lockDevice;   /* attempt to lock the device */
   unsigned long lock_timeout; /* time to wait on a lock */
   string        device<>;     /* name of device */
   };
struct Create_LinkResp {
   Device_ErrorCode  error;
   Device_Link       lid;
   unsigned short    abortPort;    /* for the abort RPC */
   unsigned long     maxRecvSize;  /* specifies max data size in bytes
                                      device will accept on a write */
};
Create_LinkResp create_link(Create_LinkParms) = 10;
```

RULE B.6.3:

To successfully complete a *create_link* RPC, a *network instrument* server **SHALL**:

1. If *lockDevice* is set to true, acquire the lock for the device.
2. Return in *lid* a link identifier to be used with future calls. The value of *lid* **SHALL** be unique for all currently active links within a *network instrument* server.
3. Return in *maxRecvSize* the size of the largest *data* parameter the *network instrument* server can accept in a *device_write* RPC. This value **SHALL** be at least 1024.
4. Return in *asyncPort* the port number for asynchronous RPCs. See *device_abort*.
5. Return with *error* set to 0, no error, to indicate successful completion.

The *device* parameter is a string which identifies the device for communications. See the document(s) referred to in section A.6, Related Documents, for definitions of this string.

RECOMMENDATION B.6.1:

A *network instrument* server should be able to maintain at least two separate links simultaneously over a single *network instrument* connection.

The *network instrument* client sends an identifying number in the *clientId* parameter. While this protocol requires no special behavior based on the value of *clientId*, the device may provide a local means to examine its value to help a user identify communication problems.

RULE B.6.4:

The *network instrument* server **SHALL NOT** alter its function based on the *clientId*.

RULE B.6.5:

If *create_link* is called when another link is not available, *create_link* **SHALL** terminate and set *error* to 9.

RULE B.6.6:

The operation of *create_link* **SHALL** ignore locks if *lockDevice* is false.

RULE B.6.7:

If *lockDevice* is true and the lock is not freed after at least *lock_timeout* milliseconds, *create_link* **SHALL** terminate without creating a link and return with *error* set to 11, device locked by another link.

RULE B.6.8:

The execution of *create_link* **SHALL** have no effect on the state of any device associated with the *network instrument* server.

OBSERVATION B.6.2:

A *create_link* RPC cannot be aborted since a valid link identifier is not yet available. A *network instrument* client should set *lock_timeout* to a reasonable value to avoid locking up the server.

Table B.4 lists *create_link error* values.

error	Meaning
0	no error
1	syntax error
3	device not accessible
9	out of resources
11	device locked by another link
21	invalid address

Table B.4 *create_link error* Values

B.6.2. destroy_link

The *destroy_link* call is used to close the identified link. The *network instrument* server recovers resources associated with the link.

```
Device_Error destroy_link (Device_Link)       = 23;
```

RULE B.6.9:

To successfully complete a *destroy_link* RPC, a *network instrument* server **SHALL**:

1. Deactivate the link identifier and recover any resources associated with the link.
2. If this link has the lock, free the lock (see *device_lock* and *create_link*).
3. Disable this link from using the interrupt mechanism (see *device_enable_srq*).
4. Return with *error* set to 0, no error, to indicate successful completion.

RULE B.6.10:

The *Device_Link* (link identifier) parameter is compared against the active link identifiers. If none match, *destroy_link* **SHALL** terminate and set *error* to 4.

OBSERVATION B.6.3:

After a *destroy_link*, the *network instrument* server typically becomes ready to execute a new *create_link*, assuming the resources have not already been utilized.

RULE B.6.11:

The execution of *destroy_link* **SHALL** have no effect on the state of any device associated with the *network instrument* server.

RULE B.6.12:

The operation of *destroy_link* **SHALL NOT** be affected by *device_abort*.

Table B.5 lists *destroy_link error* values.

error	Meaning
0	no error
4	invalid link identifier

Table B.5 *destroy_link error* Values

B.6.3. device_write

The *device_write* RPC is used to write data to the specified device.

```
struct Device_WriteParms {
    Device_Link     lid;         /* link id from create_link */
    unsigned long   io_timeout;  /* time to wait for I/O */
    unsigned long   lock_timeout /* time to wait for lock */
    Device_Flags    flags;
    opaque          data<>;   /* the data length and the data itself */
};
struct Device_WriteResp  {
    Device_ErrorCode   error;
    unsigned long      size; /* Number of bytes written */
};
Device_WriteResp device_write(Device_WriteParms)  = 11;
```

OBSERVATION B.6.4:

Note that the *opaque data*<> is not truly opaque, but is used directly by the device. The *opaque* type is used to avoid the overhead associated with character data (8 bits being promoted to 32 bits for XDR). *data* can contain up to 2^{32}-1 bytes.

The *network instrument* server has indirect control over the maximum size of *data* through the value of *maxRecvSize* returned in *create_link*.

RULE B.6.13:

To a successfully complete a *device_write* RPC, the *network instrument* server **SHALL:**

1. Transfer the contents of *data* to the device.
2. Return in *size* parameter the number of bytes accepted by the device.
3. Return with *error* set to 0, no error.

RULE B.6.14:

If the end flag in *flags* is set, then an END indicator **SHALL** be associated with the last byte in *data*.

OBSERVATION B.6.5:

If a controller needs to send greater than *maxRecvSize* bytes to the device at one time, then the *network instrument* client makes multiple calls to *device_write* to accomplish the complete transaction. A *network instrument* server accepts at least 1,024 bytes in a single *device_write* call due to RULE B.6.3.

OBSERVATION B.6.6:

The value of *data.data_len* may be zero, in which case no device actions are performed.

RULE B.6.15:

The *lid* parameter is compared to the active link identifiers. If none match, *device_write* **SHALL** terminate and set *error* to 4, invalid link identifier.

RULE B.6.16:

If *data.data_len* is greater than the value of *maxRecvSize* returned in *create_link*, *device_write* **SHALL** terminate without transferring any bytes to the device and **SHALL** set *error* to 5.

RULE B.6.17:
If some other link has the lock, *device_write* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_write* **SHALL** block until the lock is free. If the flag is not set, *device_write* **SHALL** terminate and set *error* to 11, device already locked by another link.

RULE B.6.18:
If after at least *lock_timeout* milliseconds the lock is not freed , *device_write* **SHALL** terminate with *error* set to 11, device already locked by another link.

RULE B.6.19:
If after at least *io_timeout* milliseconds not all of *data* has been transferred to the device, *device_write* **SHALL** terminate with *error* set to 15, I/O timeout. This timeout is based on the entire transaction and not the time required to transfer single bytes.

OBSERVATION B.6.7:
The *io_timeout* value set by the application may need to change based on the size of *data*.

RULE B.6.20:
If the asynchronous *device_abort* RPC is called during execution, *device_write* **SHALL** terminate with *error* set to 23, abort.

RULE B.6.21:
The number of bytes transferred to the device **SHALL** be returned in *size*, even when the call terminates due to a timeout or *device_abort*.

RULE B.6.22:
If the *network instrument* server encounters a device specific I/O error while attempting to write the data, *device_write* **SHALL** terminate with error set to 17, I/O error.

Table B.6 lists *device_write error* values.

error	Meaning
0	no error
4	invalid link identifier
5	parameter error
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.6 *device_write error* Values

B.6.4. device_read

The *device_read* RPC is used to read data from the device to the controller.

```
struct Device_ReadParms {
    Device_Link    lid;         /* link id from create_link */
    unsigned long  requestSize; /* Bytes requested */
    unsigned long  io_timeout;  /* time to wait for I/O */
    unsigned long  lock_timeout;/* time to wait for lock */
    Device_Flags   flags;
    char           termChar;    /* valid if  flags & termchrset */
};
struct Device_ReadResp {
    Device_ErrorCode error;
    long             reason;    /* Reason(s) read completed */
    opaque           data<>;    /* data_len and data_val */
};
Device_ReadResp device_read(Device_ReadParms)   = 12;
```

Bit assignments for *reason* are shown in Table B.7.

Bit #	31 - 3	2	1	0
Contents	0	END	CHR	REQCNT

Table B.7 *reason* Bit Assignments

OBSERVATION B.6.8:

Note that the *opaque data*<> is not truly opaque, but is used directly by the controller. The *opaque* type is used to avoid the overhead associated with character data (8 bits being promoted to 32 bits for XDR). *data* can contain up to 2^{32}-1 bytes.

RULE B.6.23:

To successfully complete a *device_read* RPC, a *network instrument* server **SHALL:**

1. Transfer bytes into the *data* parameter until one of the following termination conditions are met:
    a. An END indicator is read. The END bit in *reason* **SHALL** be set.
    b. *requestSize* bytes are transferred. The REQCNT bit in *reason* **SHALL** be set. This termination condition **SHALL** be used if *requestSize* is zero.
    c. *termchrset* is set in *flags* and a character which matches *termChar* is transferred. The CHR bit in *reason* **SHALL** be set.
    d. The buffer used to return the response is full. No bits in *reason* **SHALL BE** set.
2. Return with error set to 0, no error, to indicate successful completion.

If more than one termination condition is valid, *reason* contains the bitwise inclusive OR of all the reasons.

OBSERVATION B.6.9:

If *reason* is not set (value of 0) and error is zero, then the *network instrument* client could issue *device_read* calls until one of the other three termination conditions is encountered.

RULE B.6.24:

The *lid* parameter is compared against the active link identifiers. If none match, *device_read* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.25:
If some other link has the lock, *device_read* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_read* **SHALL** block until the lock is free before transferring data. If the flag is not set, *device_read* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.26:
If after at least *lock_timeout* milliseconds the lock is not freed, *device_read* **SHALL** terminate with *error* set to 11, device locked by another device and *data.data_len* set to zero.

RULE B.6.27:
If the transfer takes longer than *io_timeout* milliseconds, *device_read* **SHALL** terminate with *error* set to 15, I/O timeout, *data.data_len* set to however many bytes were transferred, and *reason* set to zero.

RULE B.6.28:
If the *network instrument* server encounters a device specific I/O error while attempting to read the data, *device_read* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.29:
If the asynchronous *device_abort* RPC is called during execution, *device_read* **SHALL** terminate with *error* set to 23, abort.

RULE B.6.30:
The number of bytes transferred from the device into *data* **SHALL** be returned in *data.data_len* even when *device_read* terminates due to a timeout or *device_abort*.

Table B.8 lists *device_read error* values.

error	Meaning
0	no error
4	invalid link identifier
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.8 *device_read error* Values

B.6.5. device_readstb

The *device_readstb* RPC is used to read a device's status byte.

```
struct Device_ReadStbResp {
   Device_ErrorCode   error;     /* error code */
   unsigned char      stb;       /* the returned status byte */
};

Device_ReadStbResp device_readstb (Device_GenericParms)  = 13;
```

RULE B.6.31:

To successfully complete a *device_readstb* RPC, the *network instrument* server **SHALL:**

1. Return in the *stb* parameter the device's status byte.
2. Return with error set to 0, no error, to indicate successful completion

OBSERVATION B.6.10:

Since not all devices directly support a status byte, how this operation is executed and the semantics of the *stb* parameter depend upon the interface between the *network instrument* server and the device.

RULE B.6.32:

If a status byte cannot be returned, *device_readstb* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.33:

The *lid* parameter is compared against the active link identifiers . If none match, *device_readstb* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.34:

If some other link has the lock, the procedure examines the waitlock flag in *flags*. If the flag is set, *device_readstb* blocks until the lock is free before retrieving the status byte. If the flag is not set, *device_readstb* **SHALL** terminate and set *error* to 11, device locked by another link.

RULE B.6.35:

If after at least *lock_timeout* milliseconds the lock is not freed, *device_readstb* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.36:

If after at least *io_timeout* milliseconds the operation is not complete, *device_readstb* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.37:

If the *network instrument* server encounters a device specific I/O error while attempting to read the data, *device_readstb* **SHALL** terminate with error set to 17.

RULE B.6.38:

If the asynchronous *device_abort* RPC is called during execution, *device_readstb* **SHALL** terminate with *error* set to 23.

Table B.9 lists *device_readstb* error values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.9 *device_readstb error* Values

B.6.6. device_trigger

The *device_trigger* RPC is used to send a trigger to a device.

```
Device_Error      device_trigger (Device_GenericParms)  = 14;
```

RULE B.6.39:

To successfully complete a *device_trigger* RPC, a *network instrument* server **SHALL**

1. Send a trigger to the associated device.
2. Return with *error* set 0, no error, to indicate successful completion.

OBSERVATION B.6.11:

Since not all devices directly support a trigger, how this operation is carried out depends upon the interface between the *network instrument* server and the device.

RULE B.6.40:

If the device does not support a trigger and the *network instrument* server is able to detect this, *device_trigger* **SHALL** terminate and set *error* to 8, operation not supported.

OBSERVATION B.6.12:

IEEE 488.1 and similar interfaces may not be able to detect that the device does not support a trigger.

RULE B.6.41:

The *lid* parameter is compared against the link identifiers. If none match, *device_trigger* **SHALL** terminate and set *error* to 4, invalid link identifier.

RULE B.6.42:

If some other link has the lock, *device_trigger* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_trigger* **SHALL** block until the lock is free before sending the trigger. If the flag is not set, *device_trigger* **SHALL** terminate and set *error* to 11, device locked by another link.

RULE B.6.43:

If after at least *lock_timeout* milliseconds the lock is not freed, *device_trigger* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.44:

If after at least *io_timeout* milliseconds the operation is not complete, *device_trigger* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.45:

If the *network instrument* server encounters a device specific I/O error while sending to trigger , *device_trigger* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.46:

If the asynchronous *device_abort* RPC is called during execution, *device_trigger* **SHALL** terminate with *error* set to 23, abort.

Table B.10 lists *device_trigger error* values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.10 *device_trigger error* Values

B.6.7. device_clear

The *device_clear* RPC is used to send a device clear to a device.

```
Device_Error device_clear (Device_GenericParms)  = 15;
```

RULE B.6.47:
To successfully complete a *device_clear* RPC, a *network instrument* server **SHALL:**
1. Clear the associated device
2. Return with *error* set to zero, no error, to indicate successful completion.

OBSERVATION B.6.13:
IEEE 488.1 and similar interfaces may not be able to detect that the device does not support a clear.

OBSERVATION B.6.14:
Since not all devices directly support a clear operation, how this operation is executed depends upon the interface between the *network instrument* server and the device.

RULE B.6.48:
If the device does not support a clear operation and the *network instrument* server is able to detect this, *device_clear* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.49:
The *lid* parameter is compared against the active link identifiers. If none match, *device_clear* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.50:
If some other link has the lock, *device_clear* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_clear* **SHALL** block until the lock is free. If the flag is not set, *device_clear* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.51:
If after at least *lock_timeout* milliseconds the lock is not freed, *device_clear* **SHALL** terminate with *error* set to 11, device locked by another device.

RULE B.6.52:
If after at least *io_timeout* milliseconds the operation is not complete, *device_clear* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.53:
If the *network instrument* server encounters a device specific I/O error while attempting to clear the device, *device_clear* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.54:
If the asynchronous *device_abort* RPC is called during execution, *device_clear* **SHALL** terminate with *error* set to 23, abort.

Table B.11 lists *device_clear error* values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.11 *device_clear error* Values

B.6.8. device_remote

The *device_remote* RPC is used to place a device in a remote state wherein all programmable local controls are disabled.

```
Device_Error      device_remote (Device_GenericParms)  = 16;
```

RULE B.6.55:

To successfully complete a *device_remote* RPC, a *network instrument* server **SHALL**:

1. Place the associated device in a remote state.
2. Return with *error* set to zero, no error, to indicate successful completion.

OBSERVATION B.6.15:

Since not all devices directly support a remote state, how this operation is executed depends upon the interface between the *network instrument* server and the device.

RULE B.6.56:

If the device does not support a remote state and the *network instrument* server is able to detect this, *device_remote* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.57:

The *lid* parameter is compared against the active link identifiers. If none match, *device_remote* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.58:

If some other link has the lock, *device_remote* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_remote* **SHALL** block until the lock is free. If the flag is not set, *device_remote* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.59:

If after at least *lock_timeout* milliseconds the lock is not freed, *device_remote* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.60:

If after at least *io_timeout* milliseconds the operation is not complete, *device_remote* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.61:

If the *network instrument* server encounters a device specific I/O error while attempting to place the device in the remote state, *device_remote* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.62:

If the asynchronous *device_abort* RPC is called during execution, *device_remote* **SHALL** terminate with *error* set to 23, abort.

Table B.12 lists *device_remote error* values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.12 *device_remote error* Values

B.6.9. device_local

The *device_local* RPC is used to place a device in a local state wherein all programmable local controls are enabled.

```
Device_Error      device_local (Device_GenericParms)  = 17;
```

RULE B.6.63:

To successfully complete a *device_local* RPC, a *network instrument* server **SHALL:**

1. Place the associated device in a local state.
2. Return with *error* set to zero, no error, to indicate successful completion.

OBSERVATION B.6.16:

Since not all devices directly support a local state, how this operation is executed depends upon the interface between the *network instrument* server and the device.

RULE B.6.64:

If the device does not support a local state and the *network instrument* server is able to detect this, *device_local* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.65:

The *lid* parameter is compared against the active link identifiers . If none match, *device_local* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.66:

If some other link has the lock, *device_local* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_local* **SHALL** block until the lock is free. If the flag is not set, *device_local* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.67:

If after at least *lock_timeout* milliseconds the lock is not freed, *device_local* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.68:

If after at least *io_timeout* milliseconds the operation is not complete, *device_local* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.69:

If the *network instrument* server encounters a device specific I/O error while attempting to place the device in the local state, *device_local* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.70:

If the asynchronous *device_abort* RPC is called during execution, *device_local* **SHALL** terminate with *error* set to 23, abort.

Table B.13 lists *device_local error* values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.13 *device_local error* Values

B.6.10. device_lock

The *device_lock* RPC is used to acquire a device's lock.

```
struct Device_LockParms {
    Device_Link       lid;          /* link id from create_link */
    Device_Flags      flags;        /* Contains the waitlock flag */
    unsigned long     lock_timeout; /* Time to wait to acquire lock */
};

Device_Error device_lock(Device_LockParms)   = 18;
```

RULE B.6.71:

To successfully complete a *device_lock* RPC, a *network instrument* server **SHALL:**

1. Acquire the device's lock.
2. Return with *error* set to zero, no error, to indicate successful completion.

RULE B.6.72:

If this link already has the lock, the *network instrument* server **SHALL** terminate with *error* set to 11, device locked by another link.

OBSERVATION B.6.17:

Multiple *network instrument* servers on the same host need to communicate with one another to implement locking since locks are global to all *network instrument* servers in a given host.

RULE B.6.73:

The *lid* parameter is compared against the active link identifiers . If none match, *device_lock* **SHALL** terminate, before trying to acquire the device's lock, with *error* set to 4, invalid link identifier.

RULE B.6.74:

If some other link has the lock, *device_lock* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_lock* **SHALL** block until the lock is free. If the flag is not set, *device_lock* **SHALL** terminate with *error* set to 11, device locked by another link.

OBSERVATION B.6.18:

The *network instrument* server blocks if another link has the lock, but does not block if another link is performing an I/O operation so long as the lock is available.

RULE B.6.75:

If after at least *lock_timeout* milliseconds the lock is not freed, *device_lock* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.76:

If the asynchronous *device_abort* RPC is called during execution, *device_lock* **SHALL** terminate with *error* set to 23, abort.

RULE B.6.77:

The locks **SHALL** be tied to the core connection between the *network instrument* client and the *network instrument* server. This means that if the *network instrument* server detects a broken connection, it **SHALL** release all of the connection's locks.

Table B.14 lists *device_lock error* values.

error	Meaning
0	no error
4	invalid link identifier
11	device locked by another link
23	abort

Table B.14 *device_lock error* Values

B.6.11. device_unlock

The *device_unlock* RPC is used to release locks acquired by the *device_lock* RPC.

```
Device_Error      device_unlock (Device_Link)  = 19;
```

RULE B.6.78:

To successfully complete a *device_unlock*, a *network instrument* server **SHALL:**

1. Release the lock.
2. Return with *error* set to zero, no error, to indicate successful completion.

RULE B.6.79:

The *Device_Link* (link identifier) parameter is compared against the active link identifiers . If none match, *device_unlock* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.80:

If this link does not have the lock, *device_unlock* **SHALL** terminate with *error* set to 12, no lock held by this link.

RULE B.6.81:

The operation of *device_unlock* **SHALL NOT** be affected by *device_abort*.

Table B.15 lists *device_unlock error* values.

error	Meaning
0	no error
4	invalid link identifier
12	no lock held by this link

Table B.15 *device_unlock error* Values

B.6.12. create_intr_chan

The *create_intr_chan* RPC is used to inform the *network instrument* server to establish an interrupt channel. See section B.3, "Interrupt Logic", for more information.

```
enum Device_AddrFamily {DEVICE_TCP, DEVICE_UDP};/* used by interrupts*/
struct Device_RemoteFunc {
    unsigned long      hostAddr;   /* Host servicing Interrupt */
    unsigned short     hostPort;   /* valid port # on client */
    unsigned long      progNum;    /* DEVICE_INTR */
    unsigned long      progVers;   /* DEVICE_INTR_VERSION */
    Device_AddrFamily  progFamily; /* DEVICE_UDP | DEVICE_TCP */
    } ;
Device_Error  create_intr_chan (Device_RemoteFunc) = 25;
```

RULE B.6.82:

To successfully complete a *create_intr_chan* RPC, a *network instrument* server **SHALL:**

1. Establish an interrupt channel to an RPC mechanism at *hostAddr* and *hostPort* whose program number is *progNum* and version is *progVers* using the underlying protocol specified in *progFamily*.
2. Return with *error* set to zero, no error, to indicate successful completion.

RULE B.6.83:

If the interrupt channel cannot be established, *create_intr_chan* **SHALL** terminate and set *error* to 6, channel not established.

RULE B.6.84:

A *network instrument* server **SHALL** support a TCP interrupt channel.

PERMISSION B.6.1:

A *network instrument* server **MAY** support a UDP interrupt channel.

OBSERVATION B.6.19:

Using UDP for the interrupt channel generally provides higher performance, but with the risks that some *device_intr_srq* RPCs might not arrive at all or that they might arrive out of order.

RULE B.6.85:

If *progFamily* is a value other than *DEVICE_TCP* or *DEVICE_UDP*, *create_intr_chan* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.86:

If the *network instrument* server does not support the protocol specified in *progFamily*, such as UDP, *create_intr_chan* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.87:

If *progNum* is not 395185, *create_intr_chan* **SHALL** terminate and set *error* to 8, operation not supported.

RULE B.6.88:

If *progVers* is not one (1), *create_intr_chan* **SHALL** terminate and set *error* to 8, operation not supported.

OBSERVATION B.6.20:

A *network instrument* client normally sets *hostAddr* equal to its own IP address.

RULE B.6.89:

If a *network instrument* server already has established the interrupt channel and it receives *create_intr_chan*, it **SHALL** perform no operation and return with *error* set to 29.

RULE B.6.90:

A *network instrument* server **SHALL** operate correctly even if an interrupt channel is not established.

Table B.16 lists *create_intr_chan error* values.

error	Meaning
0	no error
6	channel not established
8	operation not supported
29	channel already established

Table B.16 *create_intr_chan error* Values

B.6.13. destroy_intr_chan

The *destroy_intr_chan* RPC is used to inform the *network instrument* server to close its interrupt channel. See section B.3, "Interrupt Logic", for more information.

```
Device_Error  destroy_intr_chan (void) = 26;
```

RULE B.6.91:

To successfully complete a *destroy_intr_chan* RPC, a *network instrument* server **SHALL:**

1. Close its interrupt channel.
2. Return with *error* set to zero, no error, to indicate successful completion.

RULE B.6.92:

If there is no interrupt channel to be closed, *destroy_intr_chan* **SHALL** terminate with *error* set to 6, channel not established.

Table B.17 lists *destroy_intr_chan error* values.

error	Meaning
0	no error
6	channel not established

Table B.17 *destroy_intr_chan error* Values

B.6.14. device_enable_srq

The *device_enable_srq* RPC is used to enable or disable the sending of *device_intr_srq* RPCs by the *network instrument* server. See section B.3, "Interrupt Logic", for more information.

```
struct Device_EnableSrqParms   {
    Device_Link           lid;
    bool                  enable;     /* Enable or disable interrupts */
    opaque                handle<40>; /* Host specific data */
};
Device_Error  device_enable_srq (Device_EnableSrqParms) = 20;
```

RULE B.6.93:

To successfully complete a *device_enable_srq* RPC, a *network instrument* server **SHALL**:

1. Enable or disable whether this link calls *device_intr_srq* when service is requested. The interrupt for this link is disabled if *enable* is zero. It is enabled if *enable* is non-zero.
2. Store *handle<40>* so it can be passed back to the *network instrument* client in a *device_intr_srq* RPC when a service request occurs. The *network instrument* **SHALL NOT** modify this information.
3. Return with *error* set to zero, no error, to indicate successful completion.

OBSERVATION B.6.21:

A *network instrument* server maintains a service request enable state for each of its links. These states are unaffected by interrupt channel creation or destruction.

RECOMMENDATION B.6.1:

The *network instrument* client should send in the *handle* parameter a unique link identifier. This will allow the *network instrument* client to identify the link associated with subsequent *device_intr_srq* RPCs.

RULE B.6.94:

The *lid* parameter is compared against the active link identifiers . If none match, *device_enable_srq* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.95:

The *device_enable_srq* RPC **SHALL** operate the same whether or not the link has the lock.

Table B.18 lists *device_enable_srq error* values.

error	Meaning
0	no error
4	invalid link identifier

Table B.18 *device_enable_srq error* Values

B.6.15. device_docmd

The *device_docmd* RPC allows a variety of operations to be executed.

```
struct Device_DocmdParms {
    Device_Link    lid;          /* link id from create_link */
    Device_Flags   flags;        /* flags specifying various options */
    unsigned long  io_timeout;   /* time to wait for I/O to complete */
    unsigned long  lock_timeout; /* time to wait on a lock */
    long           cmd;          /* which command to execute */
    bool           network_order;/* client's byte order */
    long           datasize;     /* size of individual data elements */
    opaque         data_in<>;    /* docmd data parameters */
};
struct Device_DocmdResp {
    Device_ErrorCode error;      /* returned status */
    opaque           data_out<>; /* returned data parameter */
};
Device_DocmdResp device_docmd(Device_DocmdParms)= 22;
```

OBSERVATION B.6.22:

Note that the opaque data parameters are not truly opaque, but are used directly by the *network instrument* client and *network instrument* server. The *opaque* type is used to avoid the overhead associated with character data (8 bits being promoted to 32 bits for XDR). The data parameters can contain up to 2^{32}-1 bytes.

RULE B.6.96:

In response to a successful completion of *device_docmd*, the *network instrument* server **SHALL**:

1. Execute the operation associated with *cmd*, performing byte-swapping as necessary
2. Return in *data_out* the results defined by *cmd.*
3. Return with error set to 0, no error, to indicate successful completion

All *cmd* values are reserved for definition by related documents listed in A.6.

PERMISSION B.6.2:

In a *network instrument* server, none, some, or all values of *cmd* for *device_docmd* **MAY** be supported. The operation performed and the meaning and structure of *data_in* and *data_out* depend on the value of *cmd*.

RULE B.6.97:

The *lid* parameter is compared against the active link identifiers . If none match, *device_docmd* **SHALL** terminate with *error* set to 4, invalid link identifier.

RULE B.6.98:

The value of *cmd* is compared against the values supported by the *network instrument* server. If the particular value is not supported, *device_docmd* **SHALL** return with *error* set to 8, operation not supported.

Network_order is true (set) if the architecture of the *network instrument* client specifies byte-ordering in network order (big-endian). Network order is defined by the Internet Protocol Suite.

RULE B.6.99:

The *network instrument* server **SHALL** swap bytes in *data_in* and *data_out* as necessary if the *network instrument* server's architecture does not match that of the *network instrument* client, as indicated by the *network_order* parameter.

RULE B.6.100:

Byte-swapping **SHALL** be performed based on *datasize*, which indicates the size of individual data elements, and *data_in.data_in_len* or *data_out.data_out_len*, which indicates the total length of the data in bytes based on the following formula:

> A value of one(1) in *datasize* means byte data and no swapping is performed. A value of two(2) in *datasize* means 16-bit word data and bytes are swapped on word boundaries. A value of four(4) in *datasize* means 32-bit longword data and bytes are swapped on longword boundaries. A value of eight(8) in *datasize* means 64-bit data and bytes are swapped on 8-byte boundaries.

For example, assuming 'a', 'b', 'c', ... represent bytes, byte swapping takes place as in Table B.19, "Byte Swapping".

datasize	Data to be Swapped	Resulting Data
1	abcdefgh	abcdefgh
2	abcdefgh	badcfehg
4	abcdefgh	dcbahgfe
8	abcdefgh	hgfedcba

Table B.19 Byte Swapping

RULE B.6.101:

If some other link has the lock, *device_docmd* **SHALL** examine the waitlock flag in *flags*. If the flag is set, *device_docmd* **SHALL** block until the lock is free. If the flag is not set, *device_docmd* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.102:

If after at least *lock_timeout* milliseconds, the lock is not freed, *device_docmd* **SHALL** terminate with *error* set to 11, device locked by another link.

RULE B.6.103:

If after at least *io_timeout* milliseconds, the *cmd* cannot be completely executed, *device_docmd* **SHALL** terminate with *error* set to 15, I/O timeout.

RULE B.6.104:

If the *network instrument* server encounters a device specific I/O error while attempting the operation, *device_docmd* **SHALL** terminate with error set to 17, I/O error.

RULE B.6.105:

If the asynchronous *device_abort* RPC is called during execution, *device_docmd* **SHALL** terminate with *error* set to 23, abort.

Table B.20 lists *device_docmd error* values.

error	Meaning
0	no error
4	invalid link identifier
8	operation not supported
11	device locked by another link
15	I/O timeout
17	I/O error
23	abort

Table B.20 *device_docmd error* Values

B.6.16. device_abort

The *device_abort* RPC stops an in-progress call. This RPC is the only RPC defined by this specification which uses the abort channel between the *network instrument* client and the *network instrument* server.

```
Device_Error device_abort (Device_Link)          = 1;
```

RULE B.6.106:

To successfully complete a *device_abort* RPC, a *network instrument* server **SHALL:**

1. Initiate termination of any core channel, in-progress RPC associated with the link except *destroy_link*, *device_enable_srq*, and *device_unlock*.
2. Return with error set to 0, no error, to indicate successful completion

OBSERVATION B.6.23:

The intent of this rule is to handle the *device_abort* RPC ahead of the other operations, but due to operating system specific implementation details the timeliness cannot be guaranteed.

OBSERVATION B.6.24:

The *device_abort* RPC only aborts an in-progress RPC, not a queued RPC.

RULE B.6.107:

After replying to the *device_abort* call, the *network instrument* server **SHALL** reply to the original in-progress call which was aborted with *error* set to 23, aborted.

OBSERVATION B.6.25:

Receiving *0* on the abort call at the *network instrument* client only means that the abort was successfully delivered to the *network instrument* server.

RULE B.6.108:

The *lid* parameter is compared against the active link identifiers . If none match, *device_abort* **SHALL** terminate with *error* set to 4 invalid link identifier.

RULE B.6.109:

The operation of *device_abort* **SHALL NOT** be affected by locking.

Table B.21 lists *device_abort error* values.

error	Meaning
0	no error
4	invalid link identifier

Table B.21 *device_abort error* Values

B.6.17. device_intr_srq

A *device_intr_srq* RPC is used by a *network instrument* server to send a service request to a *network instrument* client.

```
struct Device_SrqParms {
    opaque handle<>;
};
void device_intr_srq (Device_SrqParms) = 30;
```

A *device_intr_srq* RPC is sent on the interrupt channel between a *network instrument* server and a *network instrument* client. It is used to indicate that a device is requesting service.

RULE B.6.110:

A *network instrument* server **SHALL** send *device_intr_srq* only if service requests have been enabled by *device_enable_srq* and an interrupt channel exists.

RULE B.6.111:

A *network instrument* server **SHALL** send in the *handle* parameter the data exactly as it was received in the *device_enable_srq handle* parameter for the associated link.

OBSERVATION B.6.26:

Upon receipt of *device_intr_srq*, a *network instrument* client normally takes action to service the request, but no action is required of the *network instrument* client by this specification.

C. *Network Instrument* RPCL

An ONC RPC protocol is described using RPCL. This section contains the complete listing of the protocols for the core, abort , and interrupt channels.

RULE C.1:
A *network instrument* host **SHALL** implement the following RPCL constructs.

C.1. Core and Abort Channel Protocol

```
/* Types */
typedef long Device_Link;
enum Device_AddrFamily {     /* used by interrupts */
    DEVICE_TCP,
    DEVICE_UDP
};
typedef long Device_Flags;

/*  Error types  */
typedef long Device_ErrorCode;
struct Device_Error {
    Device_ErrorCode   error;
};

struct Create_LinkParms {
    long          clientId;     /* implementation specific value */
    bool          lockDevice;   /* attempt to lock the device */
    unsigned long lock_timeout; /* time to wait on a lock */
    string        device<>;     /* name of device */
};
struct Create_LinkResp {
    Device_ErrorCode  error;
    Device_Link       lid;
    unsigned short    abortPort;   /* for the abort RPC */
    unsigned long     maxRecvSize; /* specifies max data size in bytes
                                      device will accept on a write */
};
struct Device_WriteParms {
    Device_Link       lid;          /* link id from create_link */
    unsigned long     io_timeout;   /* time to wait for I/O */
    unsigned long     lock_timeout; /* time to wait for lock */
    Device_Flags      flags;
    opaque            data<>;  /* the data length and the data itself */
};
struct Device_WriteResp  {
    Device_ErrorCode  error;
    unsigned long     size;    /* Number of bytes written */
};
struct Device_ReadParms {
    Device_Link      lid;          /* link id from create_link */
    unsigned long    requestSize;  /* Bytes requested */
    unsigned long    io_timeout;   /* time to wait for I/O */
    unsigned long    lock_timeout; /* time to wait for lock */
    Device_Flags     flags;
    char             termChar;     /* valid if flags & termchrset */
```

```
};
struct Device_ReadResp {
    Device_ErrorCode  error;
    long              reason;  /* Reason(s) read completed */
    opaque            data<>;  /* data.len and data.val */
};
struct Device_ReadStbResp {
    Device_ErrorCode  error;   /* error code */
    unsigned char     stb;     /* the returned status byte */
};
struct Device_GenericParms {
    Device_Link     lid;          /* Device_Link id from connect call */
    Device_Flags    flags;        /* flags with options */
    unsigned long   lock_timeout; /* time to wait for lock */
    unsigned long   io_timeout;   /* time to wait for I/O */
};
struct Device_RemoteFunc {
    unsigned long    hostAddr;      /* Host servicing Interrupt */
    unsigned short   hostPort;      /* valid port # on client */
    unsigned long    progNum;       /* DEVICE_INTR */
    unsigned long    progVers;      /* DEVICE_INTR_VERSION */
    Device_AddrFamily   progFamily; /* DEVICE_UDP | DEVICE_TCP */
};
struct Device_EnableSrqParms    {
    Device_Link           lid;
    bool                  enable;     /* Enable or disable interrupts */
    opaque                handle<40>; /* Host specific data */
};
struct Device_LockParms {
    Device_Link    lid;           /* link id from create_link */
    Device_Flags   flags;         /* Contains the waitlock flag */
    unsigned long  lock_timeout;  /* time to wait to acquire lock */
};
struct Device_DocmdParms {
    Device_Link    lid;           /* link id from create_link */
    Device_Flags   flags;         /* flags specifying various options */
    unsigned long  io_timeout;    /* time to wait for I/O to complete */
    unsigned long  lock_timeout;  /* time to wait on a lock */
    long           cmd;           /* which command to execute */
    bool           network_order; /* client's byte order */
    long           datasize;      /* size of individual data elements */
    opaque         data_in<>;     /* docmd data parameters */
};
struct Device_DocmdResp {
    Device_ErrorCode   error;       /* returned status */
    opaque             data_out<>;  /* returned data parameter */
};

program DEVICE_ASYNC{
    version DEVICE_ASYNC_VERSION {
       Device_Error      device_abort (Device_Link)           = 1;
     } = 1;
} = 0x0607B0;

program DEVICE_CORE {
  version DEVICE_CORE_VERSION {
    Create_LinkResp    create_link        (Create_LinkParms)      = 10;
    Device_WriteResp   device_write       (Device_WriteParms)     = 11;
    Device_ReadResp    device_read        (Device_ReadParms)      = 12;
    Device_ReadStbResp device_readstb     (Device_GenericParms)   = 13;
```

```
    Device_Error       device_trigger     (Device_GenericParms)   = 14;
    Device_Error       device_clear       (Device_GenericParms)   = 15;
    Device_Error       device_remote      (Device_GenericParms)   = 16;
    Device_Error       device_local       (Device_GenericParms)   = 17;
    Device_Error       device_lock        (Device_LockParms)      = 18;
    Device_Error       device_unlock      (Device_Link)           = 19;
    Device_Error       device_enable_srq  (Device_EnableSrqParms) = 20;
    Device_DocmdResp   device_docmd       (Device_DocmdParms)     = 22;
    Device_Error       destroy_link       (Device_Link)           = 23;
    Device_Error       create_intr_chan   (Device_RemoteFunc)     = 25;
    Device_Error       destroy_intr_chan  (void)                  = 26;
     } = 1;
} = 0x0607AF;
```

C.2. Interrupt Protocol

```
/* Types */
struct Device_SrqParms {
    opaque handle<>;
};

program DEVICE_INTR {
  version DEVICE_INTR_VERSION {
    void               device_intr_srq      (Device_SrqParms)     = 30;
     }=1;
}= 0x0607B1;
```