1394TA IICP
Draft Specification for the Instrument & Industrial Control Protocol

Draft 1.00RC3 (release candidate 3)
August 2, 1999

Sponsored by:
Instrumentation and Industrial Control Working Group (II-WG) of the 1394 Trade Association

Approved for Release by:
This document has not yet been approved for release by the II-WG or the 1394 Trade Assocation

Abstract:
This document describes a lightweight protocol for efficient asynchronous communication to electronic instrumentation and industrial control devices using the IEEE-1394 serial bus. This protocol uses a dual-duplex plug structure for transfer of data and command/control sequences. A consumer communicates to a connected producer the space available in a consumer segment buffer, so all communication is flow controlled. This document specifies the establishment, use, and maintenance of the plugs. This document also specifies the discovery process for nodes implementing the protocol, and furthermore, specifies the discovery and operation of minimal memory mapped nodes.

Keywords: protocol, instrument, industrial control, 1394, asynchronous, lightweight, flow control, discovery, memory mapped

1394 Trade Association
2350 Mission College Blvd. , Suite 350 , Santa Clara, CA, 95054, USA
http://www.1394TA.org

1394 Trade Association Specifications are developed with Working Groups of the 1394 Trade Association, a non-profit industry association devoted to the promotion of and growth of the market for IEEE 1394 computer products. Participants in working groups serve voluntarily and without compensation from the Trade Association. Most participants represent member organizations of the 1394 Trade Association. The specifications developed within the working groups represent a consensus of the expertise represented by the participants.

Use of a 1394 Trade Association Specification is wholly voluntary. The existence of a 1394 Trade Association Specification is not meant to imply that there are not other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the 1394 Trade Association Specification. Furthermore, the viewpoint expressed at the time a specification is approved and issued is subject to change, brought about through developments in the state of the art and comments received from users of the specification. Users are cautioned to check to determine that they have the latest revision of any 1394 Trade Association Specification.

Comments for revision of 1394 Trade Association Specifications are welcome from any interested party, regardless of membership affiliation with the 1394 Trade Association. Suggestions for changes in documents should be in the form of a proposed change of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally, questions may arise about the meaning of specifications in relationship to specific applications. When the need for interpretations is brought to the attention of the 1394 Trade Association, the Association initiates action to prepare appropriate responses.

Comments on specifications and requests for interpretations should be addressed to:

Editor, 1394 Trade Association
2350 Mission College Blvd. Suite 350
2350 Mission College Blvd.
Santa Clara, California 95054, USA

1394 Trade Association Specifications are adopted by the 1394 Trade Association without regard to patents which may exist on articles, materials or processes or to other proprietary intellectual property which may exist within a specification. Adoption of a specification by the 1394 Trade Association does not assume any liability to any patent owner or any obligation whatsoever to those parties who rely on the specification documents. Readers of this document are advised to make an independent determination regarding the existence of intellectual property rights which may be infringed by conformance to this specification.

Introduction

The 1394TA II-WG was formed with the following charter:

- Investigate protocols specific to instrumentation and industrial control applications.
- Efficiently and robustly transfer data in a standard way between serial bus compliant nodes.
- Use enhanced features of the IEEE1394 architecture to encapsulate existing command sets and protocols, for example GPIB.
- Expand into native IEEE1394 usage models to adopt and implement new features now possible.

The II-WG decided in September 1998 to first create a baseline document. Higher level protocol documents, for example GPIB using IICP (IICP488), will follow, building on this baseline IICP document.

Committee Membership

Chairman: Andreas Schloissnik
Company: 3A International
Email: aschloissnik@3a.com
Phone: (602) 437-1751

Secretary: Gary Sakmar
Company: Keithley Instruments
Email: gsakmar@keithley.com
Phone: (440) 542-8016

Editor: Andy Purcell
Company: Hewlett-Packard
Email: andyp@lvld.hp.com
Phone: (970) 679-5976

II-WG Reflector: '1394-ii@1394TA.org'

The following individuals are acknowledged for their contributions to this specification:

Greg Hill
Steve Schink
Dave James
Andrew Thomson

Table of contents

List of figures

List of tables

1. Instrumentation and industrial control protocol (IICP) overview

1.1 Scope

The scope of this document is to define a lightweight communication protocol for instrumentation and industrial devices using the high-speed IEEE1394-1995 serial bus. This document defines structures and efficient, flow-controlled mechanisms to transfer data and command/control sequences to devices. This document specifies the discovery of nodes that support the protocol. It describes the set up and maintenance of the connection.

It further specifies the discovery of memory mapped 1394 devices that serve as instruments or industrial devices. It defines an interrupt notification mechanism for memory mapped devices.

The scope of the baseline document does not extend to defining the content of data or command/control sequences.

The following documents are anticipated to make use of this baseline document:

1. IICP488. Specifies the use of IICP to send IEEE488 (GPIB) messages and control sequences.
2. A document (not named yet) on using IICP to communicate to a 1394-GPIB bridge device.

2. References

- IEEE Std. 1394-1995, Standard for a High Performance Serial Bus
- ANSI/IEEE Std. 1212-1994.
- IEEE 1212r Configuration ROM (approved specification or latest draft available)
- P1394a (approved specification or latest draft available)
- 1394 Open Host Controller Interface Specification, Release 1.00, October 20, 1997

3. Document definitions and notation

3.1 Word usage – shall, should, may, can

The word ***shall*** is used to indicate mandatory requirements strictly to be followed in order to conform to the specification and from which no deviation is permitted.

The word ***should*** is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited.

The word ***may*** is used to indicate a course of action permissible within the limits of the specification.

The word ***can*** is used for statements of possibility and capability, whether material, physical, or casual.

3.2 Definitions

3.2.1 connection client:
A device capable of instantiating a plug when a connection sequence is received.

3.2.2 connection manager:
A device capable of issuing a connection sequence to a connection client to cause the creation of a plug. The connection manager is responsible for sending reactivation sequences to the plug after bus resets.

3.2.3 connection register:
A 512-byte memory buffer mapped to contiguous 1394 space used for the connection lock register and connection requests and responses. The word “register” is a slight misnomer, since the connection register is much larger than a quadlet. However, this follows the naming conventions in other 1394 documents.

3.2.4 connection sequence:
A sequence of 1394 packets that cause an IICP plug to be created.

3.2.5 consumer:
A device that receives data from a producer.

3.2.6 consumer segment buffer:
Consumer memory dedicated to receiving frames from a producer. This memory is mapped to 1394 space.

3.2.7 control path (control port):
A path for control, interrupt, trigger, and commands. Actual use is determined by a higher level protocol.

3.2.8 data path (data port):
A path for data. A high level protocol using IICP plugs may use the data path for “pure” data, keeping it free of control, interrupt, trigger, etc. information.

3.2.9 frame:
A frame is a logically complete sequence of bytes written by a producer to a consumer.

3.2.10 large frame:
A large frame is a frame that is not transmitted as a small frame. A large frame is sent to 1394 space specified by the LargeFramePageTableElement registers.

3.2.11 LargeFrameConsumer register
A consumer-resident 32-bit register that a producer updates when the large frame space has been filled or the producer has sent the last content of the large frame.

3.2.12 LargeFramePageTableElement[] register array
A producer-resident array of PageTableElement registers describing the size and location of a consumer segment buffer for receiving large frame content.

3.2.13 LargeFrameProducer register:
A producer-resident 32-bit register that a consumer updates when ready to receive more large frame content.

3.2.14 node_ID:
The 10-bit bus_ID and automatically assigned 6-bit physical_ID, as defined in IEEE1394-1995, 2.2.51.

3.2.15 OHCI:
Open Host Controller Interface. This interface defines a standard set of 1394 link chip registers and the operation of a 1394 link chip.

3.2.16 PageTableElement:
A producer-resident 64-bit register containing a 16-bit length and 48-bit address. The length and address identify the size and location of consumer segment buffer space.

3.2.17 plug:
A data structure and associated software mechanisms for data and control communications between two nodes.

3.2.18 port:
There are 2 ports in an IICP plug. One port consists of the producer and consumer function for the data path. The other port consists of the producer-consumer function for the control path.

3.2.19 producer:
A device that writes data to a consumer.

3.2.20 ProducerLimits register:
A producer-resident 32-bit register that enables a consumer to control the maximum size of 1394 write transactions.

3.2.21 small frame:
A frame of size less than or equal to 512 bytes sent in one write block packet. A small frame is sent to the small frame space specified by the SmallFramePageTableElement register.

3.2.22 SmallFrameConsumer register
A consumer-resident 32-bit register that a producer updates when the next small frame does not fit in the remaining small frame buffer space or the producer has sent the maximum number of small frames allowed.

3.2.23 SmallFramePageTableElement
A producer-resident PageTableElement register describing the size and location of a consumer segment buffer for receiving small frames.

3.2.24 SmallFrameProducer register
A producer-resident 32-bit register that a consumer updates when ready to receive more small frames.

3.2.25 unique_ID:
A 64-bit concatenation of the node_vendor_id, chip_id_hi, and chp_id_lo values in the bus_info_block. See IEEE1394-1995, section 8.3.2.5.4. This is also sometimes referred to as an EUI, EUI-64, or Extended Unique Identifier.

3.3 Numeric notation

Number formats are as described in IEEE1394-1995, section 1.6.4.

3.4 State machine notation

State machines are shown in tabular format rather than the style in IEEE1394-1995.

State	Transi-tion	Condition Action	New state
State: [Name] [Actions taken upon entry]	TX#a	1st condition to be tested Actions taken when 1st condition tests true	New state
	TX#b	2nd condition to be tested Actions taken when 2nd condition tests true	New state
	…	… …	…

Figure 1 -- State machine notation

3.4.1 State machine logic

The logic for many of the conditions and associated actions are written in 'C'-style syntax. It is assumed readers of this document are familiar with 'C' syntax. The conditions shall be evaluated from top to bottom as indicated.

3.4.2 State machine transitions – text description

Following each state machine table, there is additional text that further clarifies each transition. The state machine table and state machine textual descriptions together define the state machine behavior.

3.5 Packets with data payload

There are figures in the document that show 1394 packet data payload contents. Most figures that contain a data payload are shown in short form, without the 1394 header and without the data CRC at the end of the data. To illustrate, the long form of showing a block write is shown below. Refer to the IEEE1394 specification for a description of the write-block fields.

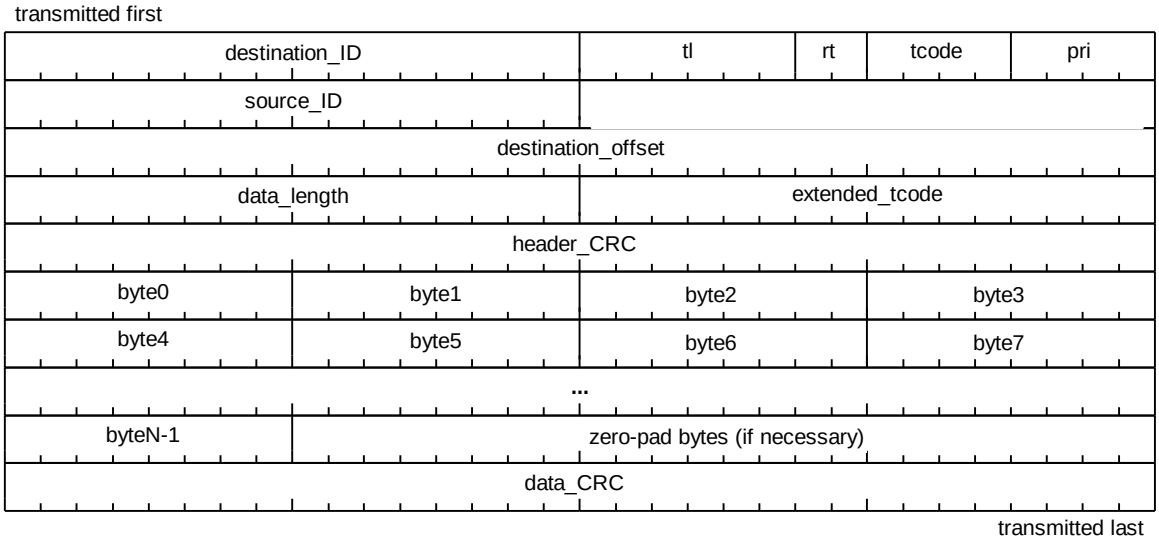

Figure 2 -- 1394 block write with header and payload

The equivalent short form is shown below.

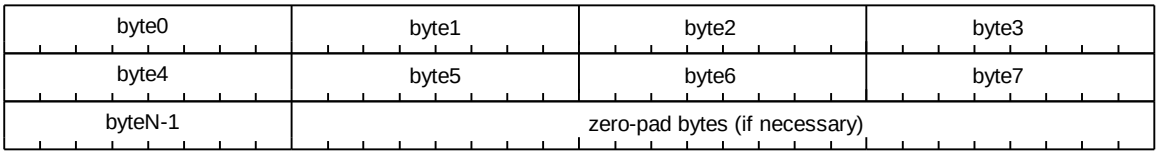

Figure 3 -- 1394 block write (short form)

3.6 Reserved fields

Some locations in a packet may be marked as ***reserved, res,*** or ***r.*** These fields are reserved for future standardization uses and shall be zero-valued. An implementation is not required, and is not expected to check all reserved fields for zero-values.

3.7 Figures depicting 1394 address space

Any figure depicting 1394 space is depicted with the convention that the 1394 space goes from lower 1394 address space (top of figure) to higher 1394 address space (bottom of figure).

Figure 4 -- Figures depicting 1394 address space

4. Functional discovery

Functional discovery is the process whereby one node on the 1394 bus discovers the attributes of another node on the 1394 bus. Node attributes are stored in the node configuration ROM.

4.1 Configuration ROM

This chapter describes and defines a method for querying an IEEE1394 node and determining its capabilities and available functionality. It is based on the IEEE 1212-1994 specification for Control and Status Register Architecture and on work currently underway in the IEEE 1212r working group for unit function discovery. The data structures, key and value types defined in this document pertain to nodes that are compliant with the IICP specification.

4.1.1 Configuration ROM structure

All IICP nodes shall provide a configuration ROM located at a fixed destination offset of FFFF F000 0400_{16}. All IICP nodes shall implement the general ROM format. All IICP nodes shall include:

- A bus information block
- A root directory
- At least one unit directory
- A text leaf containing a string for the manufacturer
- A text leaf containing a string for the unit model.

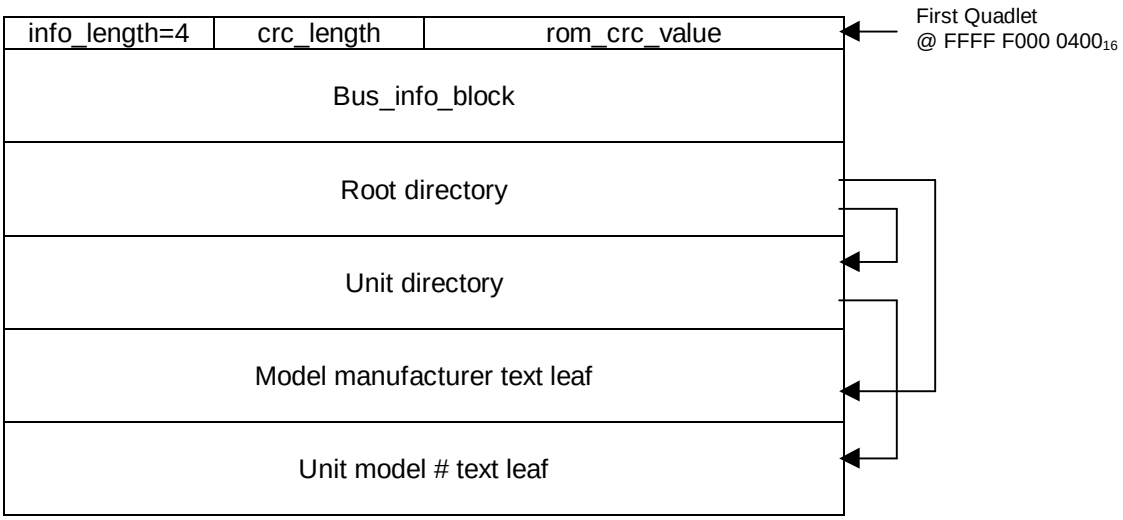

Figure 5 -- IICP configuration ROM format (one unit directory)

4.1.1.1 First quadlet

The first quadlet in the configuration ROM contains the ***info_length***, ***crc_length***, and ***rom_crc_value***. This is described in IEEE 1212-1994. Implementations shall adhere to newer applicable standards when approved. At this time, the IEEE 1212r working group recommends the ***crc_length*** to be the length in quadlets of the Bus_info_block.

4.1.1.2 Bus_info_block

The bus information block for 1394 nodes is defined in IEEE1394-1995, section 8.3.2.5.4. Implementations shall adhere to newer applicable standards (for example: P1394a) as they are approved.

4.1.1.3 Root directory

The root directory is the top level in a hierarchy of subdirectories, leaves and immediate data values. Each quadlet entry in the root directory may represent an immediate value (the quadlet contains the data value) or an indirect offset (the quadlet contains an offset value, used to create a pointer to the data). The root directory is required by IEEE1394-1995, 8.3.2.5.5, to contain module_vendor_id, node_capabilities, and node_unique_id entries. However, at this time, the node_unique_id leaf only contains redundant information, is not used by enumeration software, and the IEEE 1212r working group is indicating the node_unique_id entry is obsolete. An example root directory is shown below.

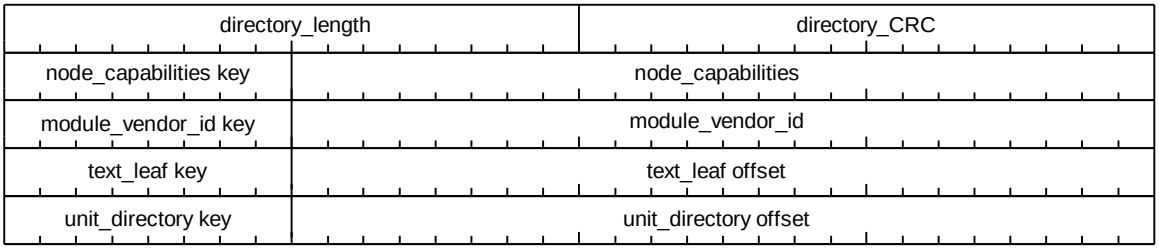

Figure 6 -- Root directory

All fields are as defined in the referenced documents. The text_leaf offset is the offset in quadlets, from the current quadlet, to a text leaf that contains a string of human-readable characters for module_vendor_id.

4.1.1.4 Unit directory

Unit directories contain additional information about a unit. Unit directories are referenced from the root directory and from any optional instance directories. Information contained in the unit directory specifies the protocol the unit uses for communications. If a node complies with more than one protocol specification, there will be multiple unit directories. There shall be at least one unit directory in an IICP compliant node.

Some IICP units are capable of generating interrupt packets and sending those packets to an interrupt handler node. The interrupt_enable_reg and interrupt_handlr_reg entries are optional entries that shall both be present for those IICP units that support this capability. If a unit does not support this capability, neither of these registers shall be present in the unit directory.

An example unit directory is shown in the figure below. Key values and macros are found in Table 1 and Table 2.

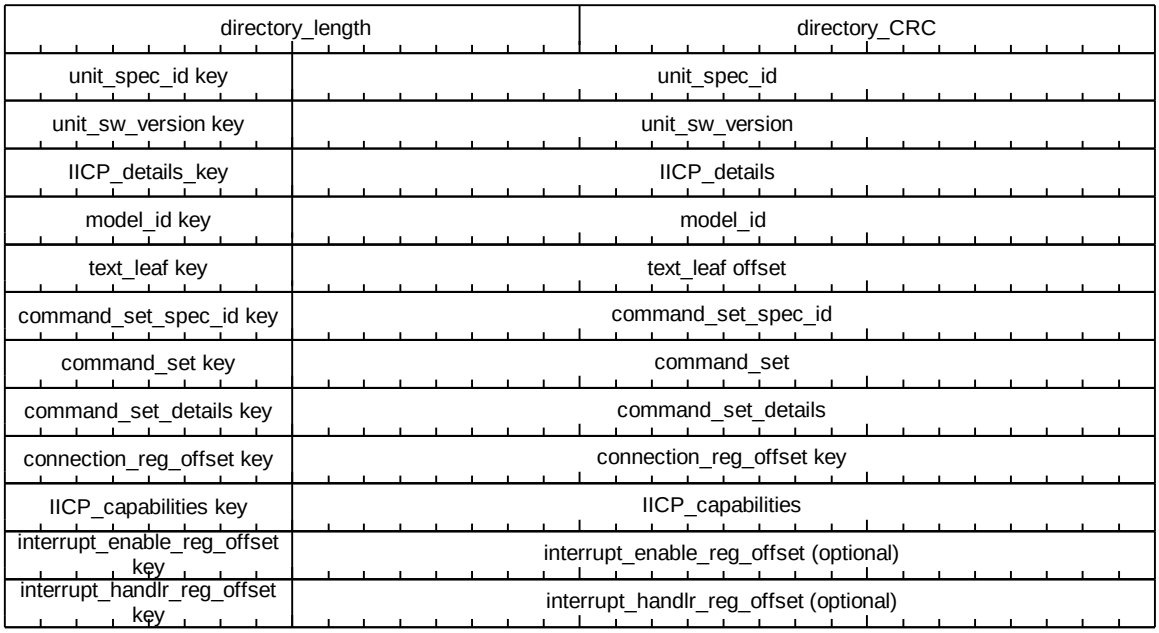

Figure 7 – IICP unit directory

- The required ***unit_spec_id*** is an immediate entry that identifies the organization that has specified the protocol for this unit. For IICP, and higher level protocols above IICP developed within the 1394 Trade Association, this shall be the number assigned to the 1394 Trade Association, 1394TA_SPEC_ID.
- The required ***unit_sw_version*** is an immediate entry that identifies the protocol. This value is determined by the organization specified in the unit_spec_id. The unit_sw_version is IICP_UNIT_SW_VERSION. This number shall represent the baseline IICP protocol and may, in some cases, result in a driver for IICP being layered on top of an operating system 1394 driver. Note that the API for an IICP driver is beyond the scope of this specification.
- The required ***IICP_details*** immediate entry specifies a revision number and details of the IICP implemented. The format of the 24 bits is shown below. The revision is interpreted as AB.CD. Decimal revision 2.39 would have nibbles ABCD valued as 0_{16}, 2_{16}, 3_{16}, and 9_{16} respectively. Each nibble shall be encoded as a binary-coded-decimal value: 0_{16} <= nibble value <= 9_{16}. The revision following 1.39 would be 1.40. IICP_details shall be set to IICP_DETAILS.

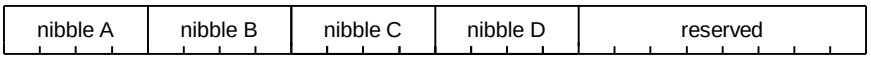

Figure 8 -- IICP_details

- The required ***model_id*** is a 24-bit immediate entry. It is the model designation assigned by the vendor.
- The required ***text_leaf offset*** is an immediate entry that specifies the offset to a text leaf that contains a textual descriptor for the previous model_id entry.
- The required ***command_set_spec_id*** is an immediate entry in the unit directory that identifies the organization responsible for the command_set definition for the unit. For this baseline IICP protocol, and for any protocol developed by a 1394 Trade Association working group, the 24-bit ***command_set_spec_id*** value shall be set to the value assigned to the 1394TA, 1394TA_SPEC_ID.
- The required ***command_set*** immediate entry that, in combination with the command_set_spec_id, specifies the command set or higher level protocol implemented by the unit. For a unit that implements IICP only, and no protocol on top of IICP, command_set shall be set to IICP_UNIT_SW_VERSION.

- The required ***command_set_details*** immediate entry specifies a revision number and details of the command_set. The format of the 24 bits is as described for IICP_details above. For a unit that implements IICP only, and no protocol on top of IICP, command_set_details shall be set to IICP_DETAILS.
- The ***connection_reg_offset*** immediate entry specifies a 1394 destination offset for the 512-byte connection register. All IICP units that are capable of performing as a connection manager or as a connection client shall have a connection_reg_offset entry. An octlet (8 bytes) is reserved at the beginning of the space for a connection lock register. It is called a lock register because connection managers must do a successful compare & swap lock request on the connection lock register before sending a connection request packet. The remaining space is used for connection requests and connection responses. The 24-bit connection_reg_offset field shall contain the offset for the connection register, in quadlets, from the base destination offset of initial register space, FFFF F000 0000_{16}.

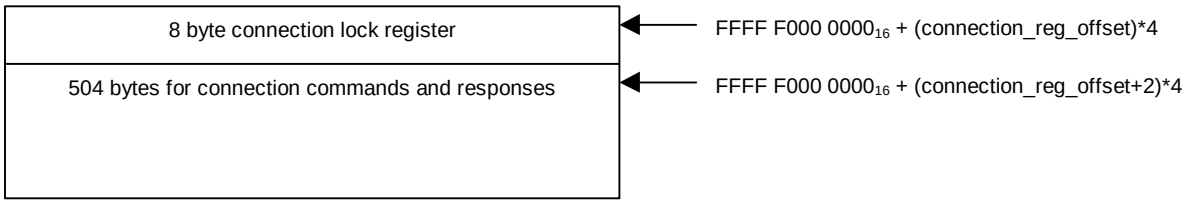

Figure 9 -- Connection register In 1394 space

A zero-valued lock register corresponds to an unlocked condition. A non-zero valued lock register corresponds to a locked condition.

Connection lock registers shall be initialized to 0000 0000 0000 0000_{16} at power-on and after a bus reset.

Only 16 byte data_length compare & swap lock requests are permitted on the 8 byte connection lock register. Only write requests are permitted on the 504 byte connection command/response space.

The connection lock register provides robustness in multi-controller and multi-threaded environments. It also simplifies device software since the device only needs to handle connection requests from at most one connection manager.

- The format of the required ***IICP_Capabilities*** immediate entry is shown below.

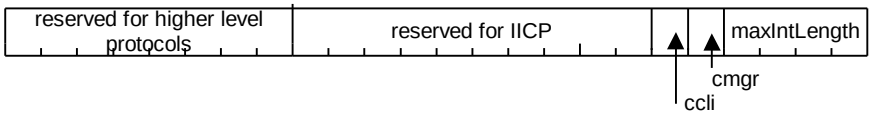

Figure 10 -- IICP_Capabilities entry

- The **reserved for higher level protocols** field is 0 unless specified in a higher-level protocol document.
- The **reserved for IICP** field is 0.
- The ***ccli*** bit (connection client bit) shall be one-valued if the unit is capable of receiving connection sequences and creating plugs.
- The ***cmgr*** bit (connection manager bit) shall be one-valued if the unit is capable of issuing connection sequences to nodes to create plugs and connection sequences to maintain those created plugs.
- The 4-bit ***maxIntLength*** field specifies the data-payload size limitations for an individual, single 1394 write block request that may communicate interrupt information. See 5.1 for further explanation. This field shall be ignored if the unit directory does not contain both

the optional interrupt_enable_reg_offset register and the optional interrupt_handlr_reg_offset register. The maxIntLength value is restricted to a value <= 8, so the maximum sized write block request containing interrupt information is 512 bytes. If the maxIntLength value is 0, this node does not support this interrupt mechanism.

$$\texttt{PayloadSizeInBytes} = 2^{\texttt{maxIntLength+1}}$$

- The optional ***interrupt_enable_reg_offset*** immediate entry contains the offset for the interrupt_enable register, in quadlets, from the base destination offset of initial register space, FFFF F000 0000_{16}. This entry is required if the interrupt_handlr_reg_offset entry is present. See section 5.1 for a detailed description of the interrupt_enable register.
- The optional ***interrupt_handlr_reg_offset*** immediate entry shall contain the offset for the 64-bit interrupt_handlr register, in quadlets, from the base destination offset of initial register space, FFFF F000 0000_{16}. The interrupt_handlr register contains the nodee_ID and the 1394 destination offset that the interrupter should use when sending an interrupt packet. This entry is required if the interrupt_enable_reg_offset entry is present. See section 5.1 for a detailed description of the interrupt_handlr register.

4.1.1.5 Configuration ROM spec_id's, unit_sw_version, and command_set values

Configuration ROM constant	Where used	Value (Hex)
1394TA_SPEC_ID	• unit_spec_id, • command_set_spec_id	$00A02D_{16}$
IICP_UNIT_SW_VERSION	• unit_sw_version	$4B661F_{16}$
IICP_DETAILS	• IICP_details, • command_set_details (if device does not implement a protocol on top of IICP)	This value is the version of IICP on the title page of the IICP document that correlates to the implementation.

Table 1 -- Configuration ROM constants

4.1.1.6 Configuration ROM key values

The unit directory for an IICP compliant node requires several entries and associated key values not defined in IEEE 1212. The key values for these entries are from the vendor-dependent key space defined in IEEE 1212. The table below summarizes the IICP entries and key values.

Configuration ROM key	Key = (key_type << 6) || key_value
IICP_details key	38_{16}
command_set_spec_id key	39_{16}
command_set key	$3A_{16}$
command_set_details key	$3B_{16}$
connection_reg_offset key	$3C_{16}$
IICP_capabilities key	$3D_{16}$
interrupt_enable_reg_offset key	$3E_{16}$
interrupt_handlr_reg_offset key	$3F_{16}$

Table 2 -- Configuration ROM key values

Note that these keys are the vendor-defined keys and that the IEEE 1212r working group is defining a method to extend the key space.

4.1.1.7 Text leaves

The format for text leaves is shown in IEEE 1212-1994, 8.2.5. Text leaves shall conform to newer standards as they are approved.

4.1.1.8 Instance directories

Instance directories are optional. If instance directories are implemented in an IICP node, the II-WG should be consulted for keywords. The table below shows the keywords the II-WG has submitted at this time.

ACTUATOR	AMPLIFIER	ANALOG_INPUT	ANALOG_OUTPUT
ANALYZER	COUNTER	DIGITAL_INPUT	DIGITAL_OUPUT
DMM	FUNCTION_GENERATOR	LINE_MONITOR	LOGIC_ANALYZER
MANOMETER	OSCILLOSCOPE	PATTERN_GENERATOR	POWER_METER
POWER_SUPPLY	RECORDER	SENSOR	SIGNAL_ANALYZER
SPECTRUM_ANALYZER	SWITCH	THERMOMETER	WAVEFORM_GENERATOR

4.1.2 Multi-protocol devices

Some IICP devices may need to implement more than one protocol. For example, a device may implement IICP and may also implement an Internet protocol 1394 stack and/or an SBP-2 1394 stack.

To do this requires the configuration ROM to implement a unit directory specifically for each protocol that is supported.

4.1.3 Read operations on the configuration ROM

An implementation shall allow quadlet reads and block reads of the configuration ROM space.

4.1.4 Device aliases (nicknames) in the configuration ROM

At this time, the IEEE 1212r working group is defining a modifiable portion of the configuration ROM to be used for a device alias or nickname. A device alias can provide a human-readable description of a device. This mechanism, when implemented in non-volatile memory, offers convenient identification of devices and it is our recommendation that device aliases be implemented when the appropriate standards are in place.

5. IICP 1394 memory-mapped I/O

Some IICP implementations may support memory mapped I/O in addition to the memory mapped I/O required by the CSR Architecture, Serial Bus, and configuration ROM. The device provides some device-dependent memory that is mapped to 1394 space.

Memory mapped 1394 I/O means that the 48-bit destination offset in a 1394 packet is used to determine the memory that is accessed on the memory mapped device. For example, a digital to analog converter may define a 48-bit destination offset as the offset that contains the digital value it will convert to an analog value. The application may issue a 1394 write quadlet request to write the new digital value to be converted. The destination offset in the write-quadlet request packet is set by the application to be the understood (device-dependent) destination offset that the memory mapped device uses for new digital data.

5.1 Interrupt mechanism for IICP memory mapped devices

An interrupt mechanism is defined for memory mapped IICP (not plug-capable) implementations.

The unit directory in the configuration ROM has a pair of optional entries:

- An interrupt_enable_reg_offset entry.
- An interrupt_handlr_reg_offset entry.

If a unit directory does not have both of these entries, then this interrupt mechanism is not supported. If a unit directory has both of these entries and the configuration ROM unit directory has a non-zero valued maxIntLength field, this interrupt mechanism is supported.

5.1.1 interrupt_enable register

The format for the interrupt_enable register is shown below.

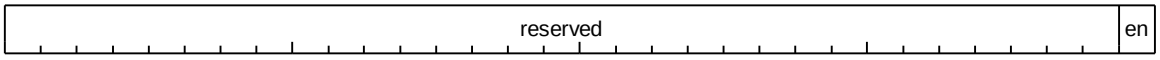

Figure 11 -- interrupt_enable register

- The ***reserved*** field shall be reserved for future use.
- The ***en-***bit (enable) enables the sending of device-dependent interrupt information to the destination_offset specified in the interrupt_handlr register. The device receiving the interrupt information must update this register and re-enable the bit (**en** = 1) to receive more interrupts. Updating the interrupt_enable register shall be done with a write quadlet request.

The initial value of the interrupt_enable register shall be 0.

After a bus reset, the interrupt_enable register en-bit shall be cleared. A device making use of this mechanism will need to update this register and set the en-bit to 1.

A read of the interrupt_enable register shall return the current register contents.

A write of this register results in an unconditional update of the contents of the register.

5.1.2 interrupt_handlr register

The format for the interrupt_handlr register is shown below.

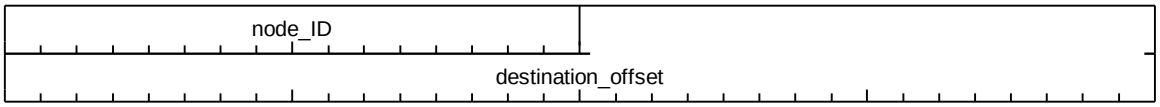

Figure 12 -- interrupt_handlr register

- The ***node_ID*** is as defined in 1394-1995, section 6.2.4.2.1. Any node utilizing this mechanism shall be responsible for updating this field after a bus reset since the node_ID may change.
- The ***destination_offset*** is used as the destination offset for the 1394 write request when sending the interrupt information to the node specified in node_ID.

The initial value of the interrupt_handlr register shall be 0.

After a bus reset, the content of the interrupt_handlr register remains unchanged. A device making use of this register may need to update the node_ID field after a bus reset.

A read of the interrupt_handlr register shall return the current register contents.

A write of this register results in an unconditional update of the contents of the register.

When an interrupt condition occurs, and the en-bit in the interrupt_enable register has been programmed to 1, the interrupting device sends device-dependent interrupt information to the destination address specified in the “interrupt_handlr register”. The size of the data payload shall not be larger than that specified in the maxIntLength field in the configuration ROM “IICP_Capabilities” entry. The content of the interrupt information is beyond the scope of this document.

The interrupt-handling device processes the interrupt and when done, re-enables the interrupting device to send another interrupt by writing to the interrupt_enable register. To re-enable, the en-bit shall be one-valued. Devices shall not send any more interrupts until after a one-value is written to the en-bit.

An implementation is not required to clear the en-bit. If an implementation does clear the en-bit, it is recommended that the en-bit be cleared prior to sending the interrupt.

5.2 1394 memory mapped only IICP device limitations

Devices equipped with only 1394 memory mapped I/O may lack certain attributes that may be key in some 1394 applications.

- Memory mapped devices may not be robust in multi-controller or multi-threaded environments.
- Memory mapped devices provide only device-dependent I/O. The memory mapped locations for I/O must be built into the software drivers.
- Memory mapped devices provide no defined data flow control mechanisms.
- There is no defined control path. If the data path becomes blocked for some reason, there is no defined mechanism for recovery.

These limitations provide motivation for the next chapter, “IICP asynchronous plug connections”.

6. IICP asynchronous plug connections

This chapter explains the creation, use, and maintenance of a communication path for the transfer of data frames and control frames from one IICP device to another IICP device.

The figure below shows a typical IICP connection between a computer and an instrument.

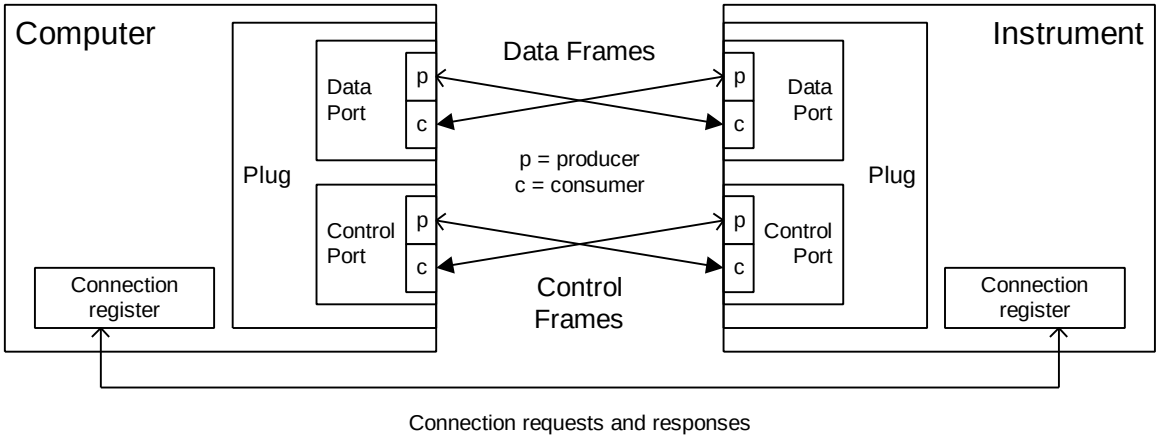

Figure 13 -- Typical IICP connection

6.1 Introduction to IICP connection plugs

A plug contains two ports: a data port and a control port.

Each port allows duplex communications with the connected node.

Using the data port, data bytes may be transferred from the computer (acting as a producer) to the instrument (acting as a consumer). Data bytes may also be transferred from the instrument (acting as a producer) to the computer (acting as a consumer).

Using the control port, control bytes may be transferred from the computer (acting as a producer of control bytes) to the instrument (acting as a control byte consumer). Control bytes may also be transferred from the instrument (acting as a producer of control bytes) to the computer (acting as a control byte consumer). Control bytes may be control messages, interrupts, triggers, and/or commands. The actual use of the data port and control ports is determined by higher-level protocols.

The control port allows the data path to remain a pure data path. This may, in some higher level protocols, free a device from parsing received packets. The control port also provides the means to communicate a potential remedy in case the data communication path hangs for some reason.

Although the IICP plug architecture allows duplex communication, simplex operation results if one of the ports does not produce frames. If an IICP connection is simplex and not duplex, resources are scaled back appropriately.

6.1.1 IICP frames

Data bytes and control bytes are transferred with one or more 1394 write operations. Data and control bytes are transferred in a logical group of bytes called a frame.

IICP plugs provide two methods, or modes, of transferring frames. ***small frame*** transfer mode is used when a frame is 512 bytes or less and fits in one 1394 write transaction. ***large frame***

transfer mode shall be used when a frame requires more than one 1394 write transaction or when a consumer disallows small frame transfer mode.

Some producers, such as a simple sensor or analog-to-digital converter may not have logical frames. Such producers are free to produce conveniently sized or optimally sized data frames. These data frames may be sized for maximum transfer efficiency. A frame should not be too small, because if transferred as a large frame, there are three 1394 transactions per frame. If transferred as a small frame, 1394 bus bandwidth is best utilized if the frame size is 512 bytes. Frames are not required to be of similar size.

Sections 6.4 and 6.5 provide an introduction to how large frame transfers and small frame transfers are accomplished. Actual producer and consumer state machine details are given later, in sections 6.20 and 6.21.

6.1.2 Plug architecture

A plug is a data structure consisting of “private” memory and “public” memory. The public memory is mapped to 1394 space and the connected node may update this mapped memory. The plug data structure is illustrated in the figure below.

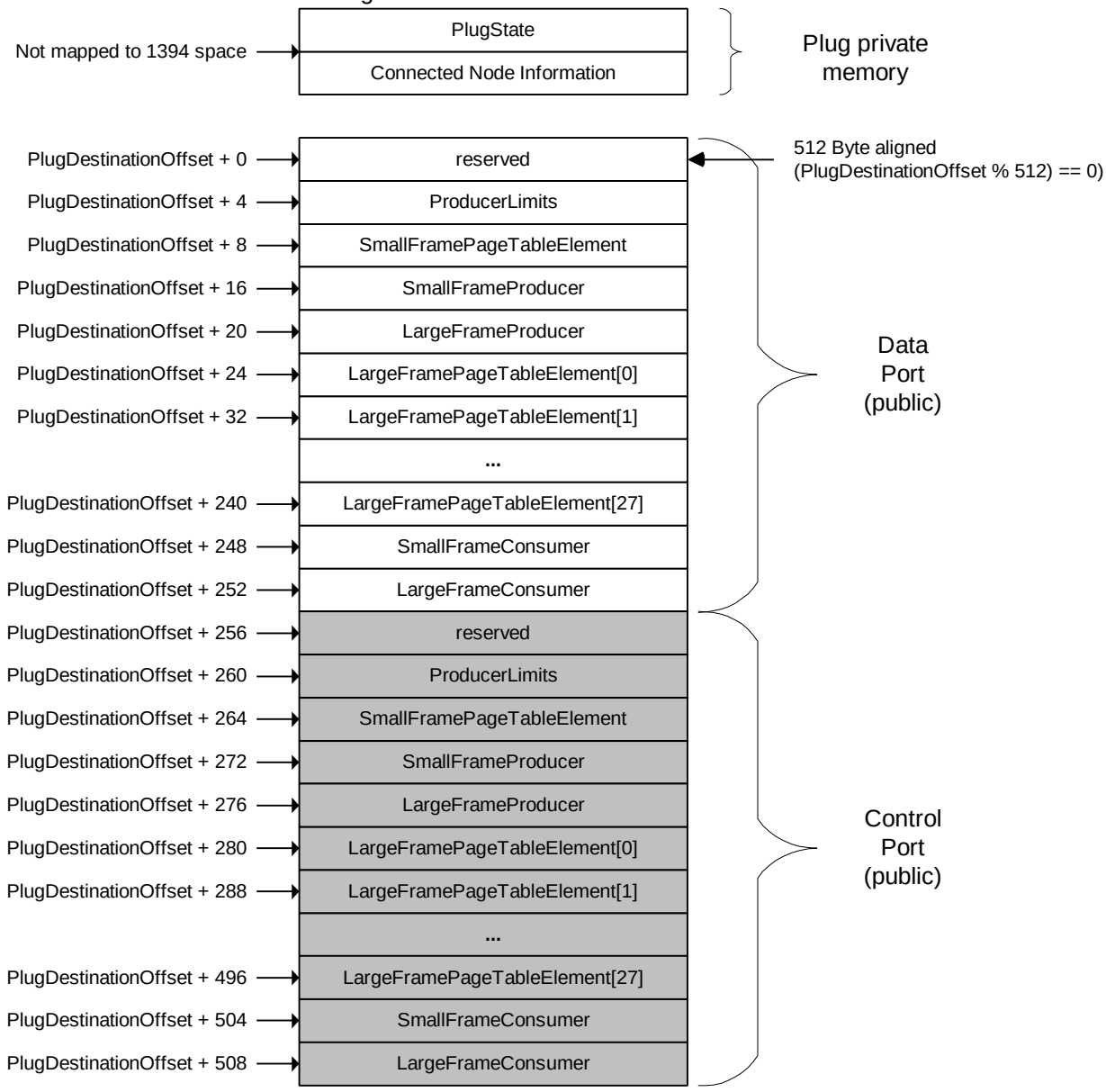

Figure 14 -- IICP plug contents

The plug private memory contains information needed in setting up and maintaining the connection. This includes plug state information and information about the connected node. It is called private because this information is not mapped into 1394 space.

PlugState information may include (but is not limited to):

- Producer state machine “state”.
- Consumer state machine “state”.

The “Connected Node” information consists of whatever information is necessary to communicate to the connected node. This information may include (but is not limited to) the following:

- The 64-bit 1394 destination address (see IEEE1394-1995, 6.2.4.2) for the plug data port on the connected node.
- The attributes about the connected node plug communicated during the connection sequence. See section 6.11.

The plug public memory is memory mapped into contiguous 1394 space. This memory contains a ProducerLimits register, a SmallFramePageTableElement register, a SmallFrameProducer register, a LargeFrameProducer register, a LargeFramePageTableElement[] register array, a SmallFrameConsumer register and a LargeFrameConsumer register.

The plug public memory shall be 512 bytes, evenly distributed to the data port and control port.

IICP implementations shall place plug public memory in 1394 space such that write transactions to public plug registers result in an interrupt to the controlling software.

Each plug shall be created with the resources shown. Higher level protocols, implemented on top of IICP, may decide how the data port and control ports are utilized.

6.2 Plug register details

This section describes the definitions of bits in each of the plug registers. The reader should skim this section first, then refer back to it when necessary.

6.2.1 ProducerLimits register

The ProducerLimits register is a 32-bit register that a consumer updates. The ProducerLimits register contains a value that limits the size of individual write requests sent by a producer. The format of the ProducerLimits register is shown below.

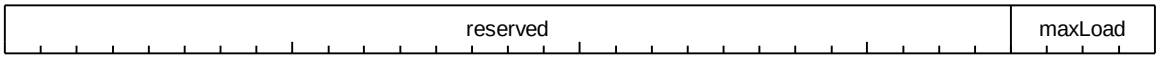

Figure 15 -- ProducerLimits register

- The 4-bit ***maxLoad*** field specifies the data-payload size limitations for individual, single 1394 segment-buffer writes, as specified in Equation 1. The amount of data in the write request cannot exceed the payloadSizeInBytes value. The maxLoad value shall be equal to or larger than 1, and is allowed to exceed the size of the node’s ROM-specified max_rec value, as defined by the Serial Bus in IEEE 1394-1995, section 8.3.2.5.4.

Equation 1: $payloadSizeInBytes = 2^{(maxLoad+1)}$

MaxLoad	PayloadSizeInBytes
1	4
2	8
…	…
10	2048 (400 Mbps maximum value)
…	

Table 3 -- maxLoad-payload values

The initial value of the ProducerLimits register shall be all zeros.

After a bus reset, the values remain unchanged.

A read of this register returns the current register contents.

A write of this register results in an update of the contents of the register. A consumer shall only update this register when the producer is expecting both a SmallFrameProducer and LargeFrameProducer update.

6.2.2 SmallFramePageTableElement register

The SmallFramePageTableElement register uses the PageTableElement format shown below. There is a 16-bit length and a 48-bit pointer to a consumer segment buffer. A consumer writes the SmallFramePageTableElement register so that the destination_offset is mapped to non-physical DMA 1394 space on the consumer. An interrupt will occur when a producer writes to the 1394 space specified in the SmallFramePageTableElement.

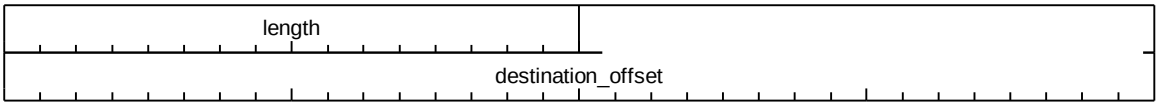

Figure 16 – PageTableElement register

- The ***length*** specifies the number of bytes a producer may write to the consumer segment buffer, which begins at the specified destination_offset. A value of 0 specifies 64 KBytes.
- The ***destination_offset*** is the 48-bit destination offset marking the start of the consumer buffer.

The initial value of the SmallFramePageTableElement register shall be all zeros.

After a bus reset, the values remain unchanged.

A read of this register returns the current register contents.

A write of this register results in an update of the contents of the register. A consumer is allowed to write this register prior to, or concurrently with, updating the SmallFrameProducer register.

6.2.3 SmallFrameProducer register

The SmallFrameProducer register is a 32-bit register that a consumer updates when ready to receive more small frames. The format of the SmallFrameProducer register is shown below.

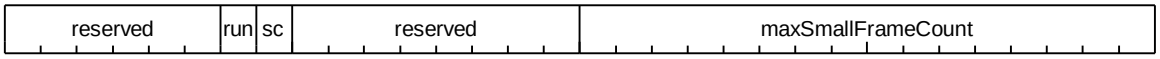

Figure 17 -- SmallFrameProducer register

- A one-valued ***run*** bit enables small frame transfers and shall be cleared when a bus reset occurs. A zero-valued ***run*** bit shall inhibit transfer of small frames. The intent is to delay small frame transfers until the consumer's state has been properly initialized.
- The ***sc***-bit is the segment count bit. The consumer, when updating the SmallFrameProducer register, shall toggle the value in the sc-bit. The first consumer update of the SmallFrameProducer register shall set the sc-bit to one, the second consumer update of the SmallFrameProducer register shall set the sc-bit to zero, and so on.
- The 16-bit ***maxSmallFrameCount*** field specifies the maximum number of small frames that a producer can send before updating the consumer SmallFrameConsumer register. This serves to limit the size of data structures the consumer may require for processing small frames. If the maxSmallFrameCount is 0, the producer shall send all frames using the large frame transfer mode and the producer shall ignore the SmallFramePageTableElement register.

The initial value of the SmallFrameProducer register shall be all zeros.

After a bus reset, the run bit is set to 0. The other bits are not changed.

A read of this register returns the current register contents.

A write of this register results in an update of the contents of the register, provided the sc-bit is a different value. The consumer is allowed to update the producer SmallFrameProducer register to initially start small frame transfers, after a bus reset, and after the producer has written to the consumer SmallFrameConsumer register and the consumer is ready for more small frames.

6.2.4 LargeFrameProducer register

The LargeFrameProducer register is a 32-bit register that a consumer updates. The consumer updates the LargeFrameProducer register when the consumer is ready to receive large frame content. The format of the register is shown below.

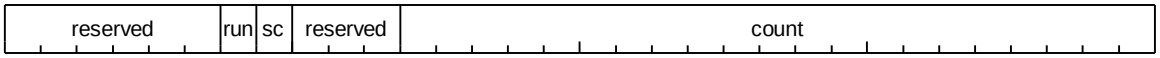

Figure 18 -- LargeFrameProducer register

- A one-valued ***run*** bit enables the operation of the producer and shall be cleared when a bus reset occurs. A zero-valued ***run*** bit shall inhibit large frame transfers.
- The ***sc***-bit is the segment count bit. The consumer, when updating the LargeFrameProducer register, shall toggle the value in the sc-bit. The first consumer update of the LargeFrameProducer register shall set the sc-bit to one, the second consumer update of the LargeFrameProducer register shall set the sc-bit to zero, and so on.
- The 21-bit ***count*** specifies the total number of bytes in the consumer segment buffer described in the LargeFramePageTableElement[] registers for receiving large frame content. The consumer segment buffer space is described beginning with PageTableElement[0].

The initial value of the LargeFrameProducer register shall be all zeros.

After a bus reset, the run bit is set to 0. The other bits are not changed.

A read of this register returns the current register contents.

A write of this register results in an update of the contents of the register, provided the sc-bit is a different value. The consumer is allowed to update the LargeFrameProducer register to initially start large frame transfers, after a bus reset, and after the producer has written to the consumer LargeFrameConsumer register and the consumer is ready for more data.

6.2.5 LargeFramePageTableElement registers

The LargeFramePageTableElement[] array consists of PageTableElement registers as shown in Figure 16. The LargeFramePageTableElement[] registers point to consumer segment buffers mapped to 1394 space on the connected node.

The multiple LargeFramePageTableElement[] registers allow a consumer to program a producer with a scatter/gather list describing the location of a consumer segment buffer in possibly non-contiguous 1394 space.

Use of scatter/gather may increase efficiencies by limiting the number of times 1394 data gets copied. For example, OHCI 1394 link implementations allow physical memory to be directly mapped to 1394 space. It is normal (at least on the computer side), for an application buffer, or user-space buffer, to reside in multiple physical memory locations. If a scatter/gather list is communicated to the producer, the producer can write 1394 data directly to the user-space buffer, filling up different physical pages in memory.

To simplify producer implementations, the consumer shall program the length field of all but the first and the last relevant LargeFramePageTableElement registers with a length = 2^N. N shall be the same value for all but the first and last relevant LargeFramePageTableElement[] registers.

The initial value of the LargeFramePageTableElement[] registers shall be all zeros.

After a bus reset, the values remain unchanged.

A read of any LargeFramePageTableElement[] registers returns the current contents.

A write of LargeFramePageTableElement[] registers results in an update of the contents of the registers. A consumer is allowed to write this register immediately prior to, or concurrently with, updating the LargeFrameProducer register.

6.2.6 SmallFrameConsumer register

The SmallFrameConsumer register is a 32-bit register that a producer updates when the small frame buffer space has been filled such that the next small frame would not fit in the remaining space or the producer has sent the maximum number of small frames. The format is shown below.

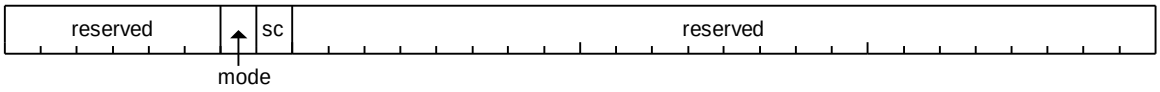

Figure 19 – SmallFrameConsumer register

- The producer shall write a one-valued (SFB_FULL) ***mode***-bit when the small frame buffer space is exhausted or the producer has sent the maximum number of small frames.
- The ***sc***-bit is the segment count bit. The producer, when updating the consumer's SmallFrameConsumer register, shall set this bit to the most recent ***sc***-bit value that the consumer wrote into the producer's SmallFrameProducer sc-bit. This bit distinctively labels the sequential handshakes between the consumer and the producer.

The initial value of the SmallFrameConsumer register shall be all zeros.

After a bus reset, the values remain unchanged.

A read of this register returns the last successfully written data.

A write of this register results in an update of the contents of the register provided the sc-bit is the same value the consumer wrote to the SmallFrameProducer register.

6.2.7 LargeFrameConsumer register

The LargeFrameConsumer register is a 32-bit register that a producer updates. Examples of when the producer updates this register include:

1) The producer has finished the transfer of a large frame.
2) There is no space left in the buffer space described by the LargeFramePageTableElement[] registers.

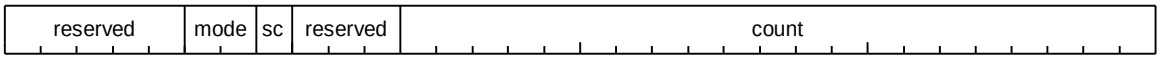

Figure 20 -- LargeFrameConsumer register

- The 2-bit ***mode*** field provides frame-completion information, as specified in the table below.

Mode	Name	Descriptionf
0	FREE	Initial (never written) state. Shall never be written by the producer.
1	MORE	Indication for leading (not end of frame, last) content.
2	LAST	Indication for last, end of frame content.
3	TRUNC	Indication for truncated frame content, end of frame

Table 4 -- LargeFrameConsumer.mode definition

- The ***sc***-bit is the segment count bit. The producer, when updating the consumer's LargeFrameConsumer register, shall set this bit to the most recent ***sc***-bit value that the consumer wrote into the producer's LargeFrameProducer.sc bit. This bit distinctively labels the sequential handshakes between the consumer and the producer.
- The 21-bit ***count*** value identifies how many bytes have been written by the producer to the buffers identified in the LargeFramePageTableElement[] registers since the last LargeFrameProducer update. Note that this value may be less than the amount of bytes the consumer allowed the producer to send.

The initial value of the LargeFrameConsumer register shall be all zeros.

After a bus reset, the values remain unchanged.

A read of this register returns the last successfully written data.

A write of this register results in an update of the contents of the register provided the sc-bit is the same value the consumer wrote to the LargeFrameProducer register.

6.3 1394 operations allowed on plug registers

Plug registers shall be updated with write block or write quadlet transactions. The destination offset in the 1394 write request shall be quadlet aligned. The length of the write transaction shall be a multiple of 4.

IEEE 1394 lock request transactions are not permitted on plug registers.

6.3.1 Efficient updating of plug registers

For efficiency, a consumer is allowed to update multiple port registers with one write block transaction. However, registers may only be updated at the proper times.

If a consumer updates producer port registers with more than one write operation, the consumer update shall affect the LargeFrameProducer only once, and the write affecting the LargeFrameProducer shall occur after, or concurrent with, the writing of the LargeFramePageTableElement[] array. The LargeFramePageTableElement[] array registers shall not be changed after a LargeFrameProducer update until the producer updates the LargeFrameConsumer register.

Similarly, the consumer update of the producer port shall affect the SmallFrameProducer only once, and the write affecting the SmallFrameProducer shall occur after, or concurrent with, the writing of the SmallFramePageTableElement. The SmallFramePageTableElement register shall not be changed after a SmallFrameProducer update until the producer updates the SmallFrameConsumer register.

6.4 Large frame transfers

Frames that do not fit the requirements for small frames are transferred using the large frame transfer mode. In this mode, LargeFrameConsumer and LargeFrameProducer registers are updated after each frame transfer.

The sequence below illustrates a producer sending large frames to a consumer. The sequence assumes the ProducerLimits register has been previously updated. Refer to the producer and consumer state machines in sections 6.20 and 6.21for more detail.

1. The consumer, when ready to receive more frame contents:
   a) Sends a write block transaction to update the producer LargeFrameProducer and LargeFramePageTableElement[] registers as necessary. After the LargeFrameProducer register is updated, the consumer is not permitted to change the LargeFramePageTableElement[] registers until after a LargeFrameConsumer update.
2. The producer:
   a) Waits for a request from an application layer to transfer frame content.
   b) Waits for a LargeFrameProducer update if necessary. This is necessary if the LargeFrameProducer register has never been updated or the producer has written to the consumer LargeFrameConsumer register.
   c) Writes the frame content to the consumer segment buffer described by LargeFramePageTableElement[0]. This may consist of several 1394 packets. If multiple LargeFramePageTableElement[] registers have been set up, the producer sequences through PageTableElement[0], PageTableElement[1], and so on.
   d) Updates the consumer's LargeFrameConsumer register. This lets the consumer know how much data the producer sent.
3. The consumer:
   a) Processes the data.

Repeat steps 1a through 3a.

The table below shows the general flow for large frame transfers.

Producer		**Consumer**
	←	Update the LargeFrameProducer and LargeFramePageTableElement[] registers.
Send data. May be multiple write block requests.	→	
Update the consumer's LargeFrameConsumer register.	→	
		Read how much data the producer sent. Process the data.
	←	Update the LargeFrameProducer and LargeFramePageTableElement[] registers.
Send data. May be multiple write block requests.	→	
Update the consumer's LargeFrameConsumer register.	→	
...	...	...

Figure 21 – Large frame transfers

6.4.1 Sequential and non-sequential writes for large frame transfers

Normally, a producer will fill the consumer buffer space sequentially. However, some consumers may not care about the ordering of writes from the producer. This is determined during the connection sequence.

If a consumer requires sequential writes, the initial write request is sent with a destination offset equal to the LargeFramePageTableElement[0] destination_offset value. Subsequent packets are sent to a destination offset increasingly offset from the first destination_offset value. The increasing offset value corresponds to the amount of data sent. These writes continue until the complete frame has been sent or the consumer buffer space specified in the LargeFramePageTableElement[0] is full. If the buffer specified in LargeFramePageTableElement[0] is full, and multiple LargeFramePageTableElement[] registers

have been set up, and there is still more frame content to be sent, writes continue in the same way, beginning with a destination offset equal to the LargeFramePageTableElement[1] destination_offset value.

If the consumer does not require sequential writes during large frame transfers, the producer is free to write the data with an unspecified ordering for the destination offsets, eventually filling up the consumer segment buffers. The result for both sequential and non-sequential writes is the same – a filled or partially filled consumer buffer space that begins with the consumer buffer space specified in LargeFramePageTableElement[0].

6.5 Small frame transfers

A small frame is a frame that may be sent in a single write transaction with payload size <= 512 bytes.

There are many situations where a computer may repeatedly send small frames to a device. An example is when a computer is sending some kind of query to an instrument to get measurement results.

For small frames, there is no need for flow control updates for every frame. Instead, the flow control updates only need occur when the next small frame does not fit completely into the small frame space. This allows frame transfers using just one 1394 transaction per frame. This is more efficient than the three 1394 transactions used in large frame mode, where LargeFrameConsumer and LargeFrameProducer registers are updated for each frame. Small frame transfers rely on 1394 drivers sending the small frame in one 1394 transaction.

The sequence below illustrates a producer sending small frames to a consumer. The sequence assumes the ProducerLimits register has been previously updated. Refer to the producer and consumer state machines in sections 6.20 and 6.21for more detail.

1. The consumer, when ready for more small frames:
   a) Updates the producer SmallFramePageTableElement and SmallFrameProducer register as necessary. After the SmallFrameProducer register is updated, the consumer is not permitted to change the SmallFramePageTableElement register until after a SmallFrameConsumer update.
2. The producer:
   a) Waits for a request from an application layer to transfer a small frame.
   b) Waits for a SmallFrameProducer register update if necessary. This is necessary if the SmallFrameProducer register has never been updated or the producer has written to the consumer SmallFrameConsumer register.
   c) Checks that the frame may be sent as a small frame.
   d) Writes the frame to the small frame space specified in the SmallFramePageTableElement register, offset by an amount equal to the amount of data transferred to the small frame space since the last SmallFrameProducer register update.
3. The consumer:
   a) Receives the small frame and forwards it to the application.

Steps 2 and 3 are repeated until the next small frame will not fit completely into the remaining small frame space or the producer has sent the maximum allowed number of small frames. When that happens, the producer shall not send any part of the small frame. The producer shall instead update the consumer's SmallFrameConsumer register, indicating the remaining small frame space is insufficient size. The consumer then updates the producer's SmallFrameProducer register when ready for more data.

The table below shows the general flow for small frame transfers.

Producer		Consumer
	←	Update the SmallFramePageTableElement and SmallFrameProducer register as necessary.
Receive transfer request. Perform tests. Send the small frame, using one 1394 transaction.	→	
		Interrupt occurs when frame received. Send frame to application.
...	...	...
Receive transfer request. Perform tests and see the small frame will not fit into the small frame space. Update the SmallFrameConsumer register.	→	
	←	Process the SmallFrameConsumer update. Update the SmallFramePageTableElement and SmallFrameProducer register as necessary.
Receive transfer request. Perform tests. Send the small frame, using one 1394 transaction.	→	
...	...	...

Figure 22 -- Small frame transfers

The size of small-frames is indicated by the 1394 header and is not replicated in the payload portion of the packet.

For this reason, implementations are expected to have hardware/software mechanisms for associating the write-transaction payload size (from the write request packet header) with the data payload.

After receiving a small frame, a consumer shall not hold off subsequent small frames via ack_busy's or any other mechanism while the small frame is being processed. A consumer is obligated to let the producer send maxSmallFrameCount small frames, so long as the small frame consumer segment buffer is not filled.

6.6 Mixing of large frame mode and small frame transfer mode

Once a producer has started a large frame transfer, small frame transfers are not permitted until the LargeFrameConsumer register has been updated with a mode value indicating the end of a large frame. Once the LargeFrameConsumer register has been updated with a mode value indicating the end of a large frame, small frame transfers may resume, beginning at an offset equal to the total size of all of the small frames sent before the large frame transfer commenced.

A producer is allowed to send a small frame and then begin a large frame transfer without updating the SmallFrameConsumer register.

A producer is allowed to send frames that fit the criteria for small frames using the large frame transfer mode.

6.7 Consumer segment buffers

A producer writes frame contents to consumer segment buffers on the connected node. A consumer may utilize one or more consumer segment buffers for receiving large frames. A consumer may utilize one or more consumer segment buffers for receiving small frames. The use of multiple segment buffers allows overlap – the consumer may process a filled segment buffer while the producer is filling up an available segment buffer. The figure below illustrates a suggested strategy of using 2 large frame consumer segment buffers and 1 small frame consumer segment buffer.

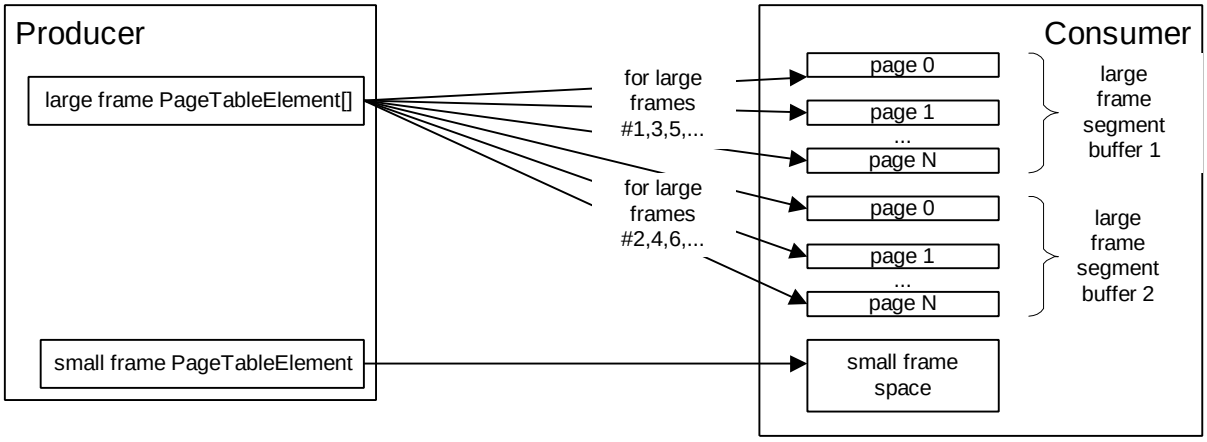

Figure 23 -- Consumer segment buffers

6.8 Plug schematics

For clarity, the schematic view of 2 plugs connected together to form a dual-duplex communication path is shown below. “p” is short for producer, “c” is short for consumer. Note that although 2 receive segment buffers are shown in the data paths, only 1 receive segment buffer is required.

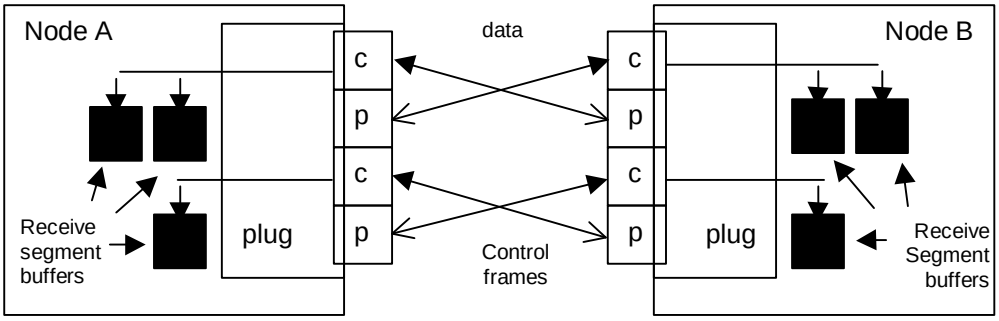

Figure 24 -- IICP plug schematic

This may also be drawn as:

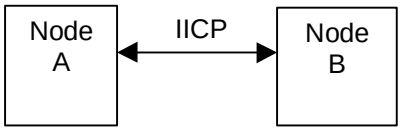

Figure 25 -- Shorthand IICP plug schematic

6.9 Multiple devices

For a node to communicate to more than one other node, separate and distinct plugs need to be created for each attached node. In the figure below, there would be N plugs on the computer for the N devices.

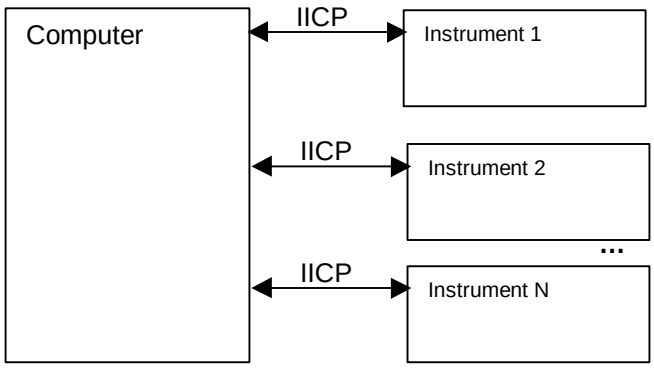

Figure 26 -- Multiple instrument connections

6.10 Connection variations

In all cases, a plug is used for data and control communications between two nodes. A plug is not expandable.

There is no limit in this specification concerning the number of plugs a device may create.

An IICP device that is plug-capable may also implement device-dependent 1394 memory mapped capabilities.

6.11 Creating an IICP connection

An IICP connection manager establishes an IICP connection by sending connection request packets to device1 and device2. In the figure below, Device1 and Device2 are attached nodes on the bus, distinct from the node acting as the IICP connection manager.

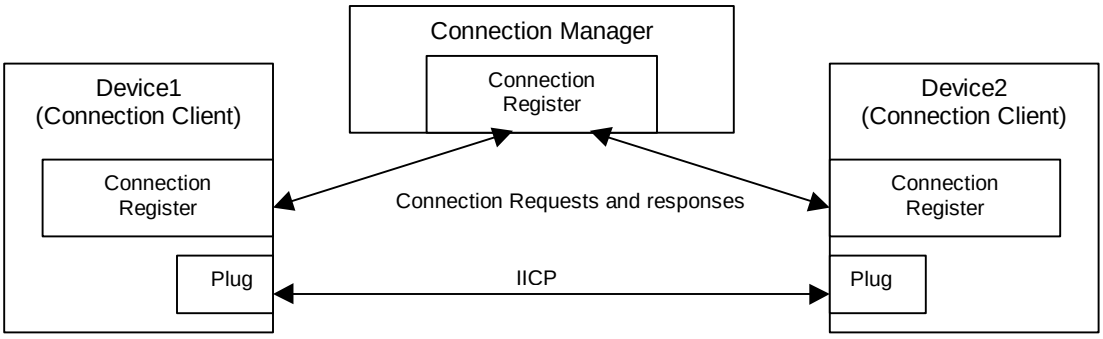

Figure 27 -- Connection manager with 2 independent devices

An IICP device may be integrated into the connection manager, as shown below. In this case, connection requests sent to Device1 and connection responses received from Device1 are not 1394 packets but rather internal software actions. Device1 is both a connection manager and a connection client.

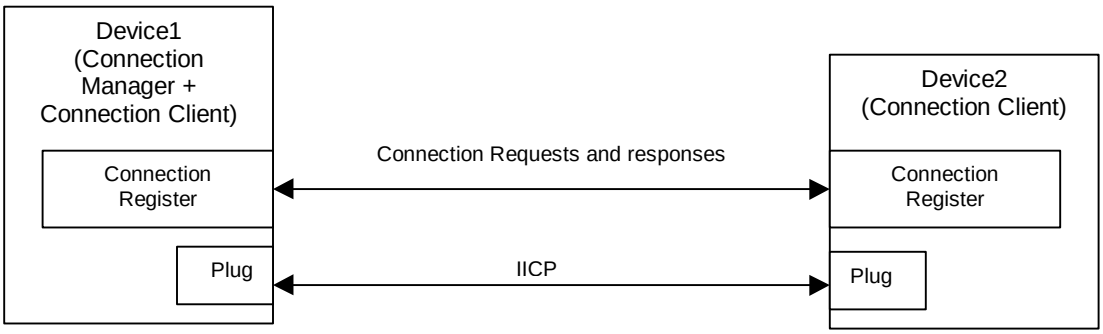

Figure 28 -- Connection manager with 1 independent device

The table below shows the general flow in creating a connection between device1 and device2. See section 6.18 for the connection manager state machine and section 6.19 for the connection client state machine for complete details.

Step	Device1		Connection Manager		Device2
1			Lock connection register.		
2		←	Lock device1 connection register.		
3			Lock device2 connection register.	→	
4		←	Send connection request packet (CREQ1) to device1.		
5	Create plug. Send connection response packet (CRESP).	→			
6			Send connection request packet (CREQ1) to device2.	→	
7				←	Create plug. Send connection response packet. (CRESP1).
8		←	Send connection request packet (CREQ2) to device1.		
9	Allocate segment buffers. Send connection response packet (STATUS).	→			
10			Send connection request packet (CREQ2) to device2.	→	
11				←	Allocate segment buffers. Send connection response packet (STATUS).
12			Unlock connection register.		
13		←	Unlock device1 connection register.		
14	Device1 may now update plug registers on device2.		Unlock device2 connection register.	→	
15					Device2 may now update plug registers on device1.

Figure 29 -- Establishing an IICP connection

6.11.1 Connection creation sequence

The details for each of the steps in the table are given below.

6.11.1.1 Connection manager locks its connection lock register

The connection manager shall first lock its connection lock register. Locking of its connection lock register is done with an atomic test and set operation. The need to specify “atomic” is to make the point that the connection manager must be able to test the state of its own lock register and then set the lock, without worrying about an external lock request from another device interfering.

6.11.1.2 Connection manager locks device1 connection lock register

The connection manager next locks the device1 connection register. Locking of all non-connection manager connection registers is accomplished by using a 1394 compare and swap lock transaction. The connection manager executes a 1394 compare & swap lock operation on the device connection lock register. Refer to IEEE1394-1995 sections 3.5.2, 6.2.2.3.2, 6.2.2.3.4, 6.2.4.9, and 7.3.4.3 for more details on compare and swap lock requests and responses. A lock request packet is shown below.

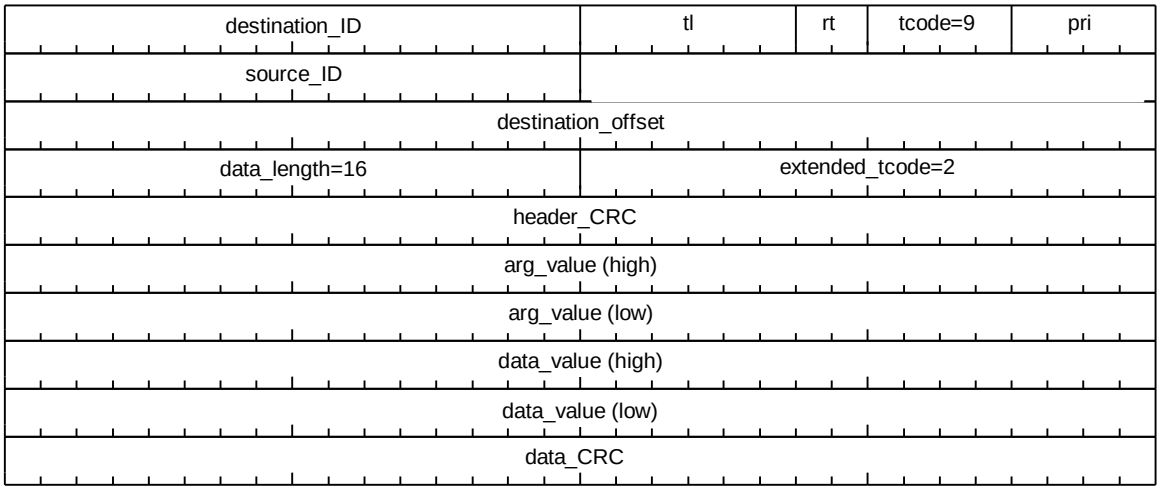

Figure 30 -- Connection register lock request packet

- ***destination_offset*** shall be set to the location of the connection register, as specified in the configuration ROM.
- ***arg_value*** shall be 64-bits and shall be all 0's when attempting to lock the connection register, since an unlocked connection register is by definition, 0-valued.
- ***data_value*** shall be 64-bits and shall be the unique_ID of the device attempting to lock the connection register.

A compare_swap lock request performs the following equivalent "C" code:

```
if ( old_value == arg_value ) new_value = data_value;
return old_value;       // in the lock response packet
```

The device returns a lock response. If the lock response indicates old_value = 0, the lock was successful. If the lock response indicates non-zero, the lock was unsuccessful. A lock response packet is shown below.

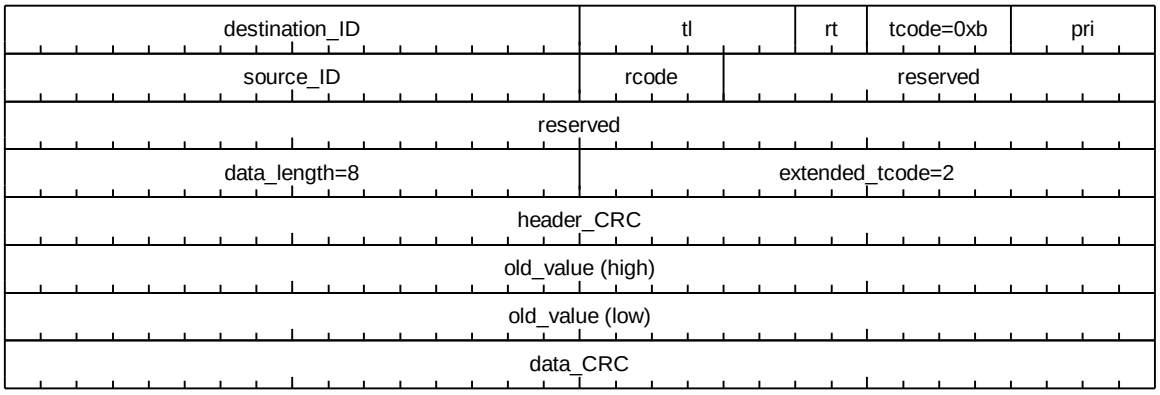

Figure 31 -- Connection register lock response packet

- ***rcode*** is the appropriate response code.
- ***old_value*** is the original value of the connection lock register. If 0, the connection register has been successfully locked.

6.11.1.3 Connection manager locks device2 connection register

The connection manager next locks the device2 connection lock register. This is again done with a compare and swap lock request operation as described in 6.11.1.2.

If successful at locking the connection manager connection lock register and all other necessary device connection lock registers, the connection manager temporarily “owns” the connection services of all the devices it has locked. If unsuccessful, the connection manager shall unlock all the connection lock registers it was successful in locking, then wait an implementation dependent period of time before retrying.

The connection manager is allowed to lock the connection registers for only a single operation. This is to prevent “hogging” of the connection registers by any one connection manager.

Connection clients maintain locks for time = CCLI_LOCK_TIMEOUT. See section 6.17 for the value. If the connection manager has not unlocked a connection register in CCLI_LOCK_TIMEOUT, the connection client shall assume some anomalous event occurred and shall unlock its own connection lock register. Any resources allocated due to received connection requests may and should be freed. If the CCLI_LOCK_TIMEOUT does occur, any further transactions (except a valid lock request) to the connection register shall fail. The device shall send a response packet as shown in Figure 35, with a status indication of CRS_REG_NOT_LOCKED.

If a bus reset occurs and a device connection register is locked, the connection register is implicitly unlocked by the device. The connection manager then re-locks connection registers as necessary.

6.11.1.4 Connection manager sends connection request packet (CREQ1) to device1

The connection manager next sends a connection request packet (CREQ1) to device1. The figure below shows the connection request packet. The destination offset for this connection request packet is the connection register location + 8 (bytes).

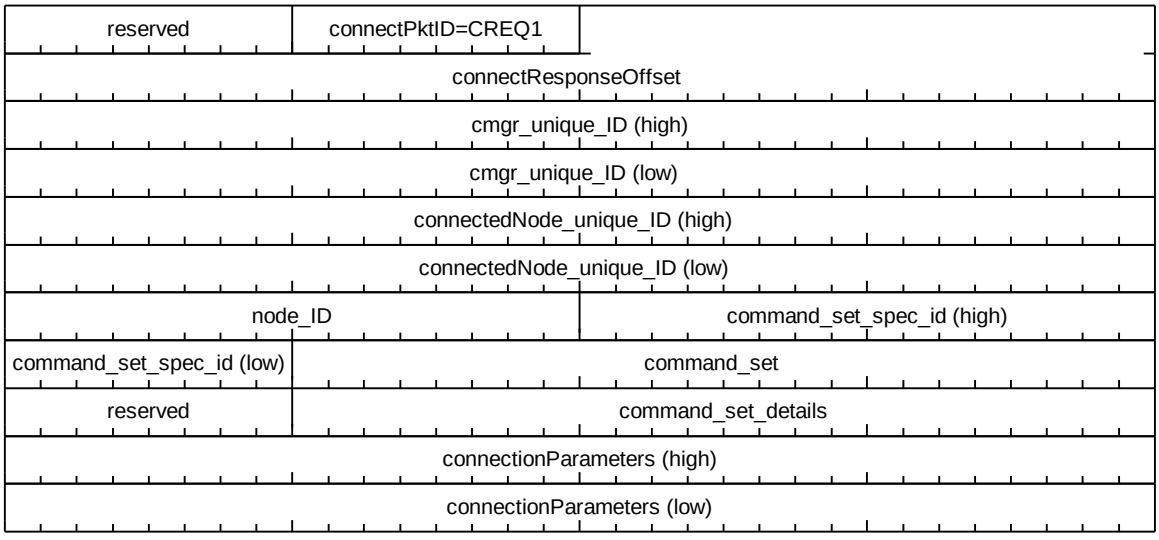

Figure 32 -- Connection request packet (CREQ1)

- ***connectPktId*** identifies the connection packet as a connection request. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 48-bit destination offset for the connection response packet.
- ***cmgr_unique_ID*** is the 64-bit unique_ID for the connection manager.
- ***connectedNode_unique_ID*** is the 64-bit unique_ID for the other device being connected.

- ***node_ID*** is the 16-bit node_ID for the other device being connected.
- ***command_set_spec_id*** specifies the organization that defines the following command_set value.
- ***command_set*** identifies the higher level protocol to be used for the connection. The command_set value should match a unit directory command_set value in the configuration ROM.
- ***command_set_details*** specifies the version of the command_set to be used.
- ***connectionParameters*** is a 64-bit field that is defined by a higher level protocol.

6.11.1.5 Device1 sends response packet (CRESP)

Device1 stores the information provided in the CREQ1 packet and formulates a connection response packet to be sent to the connection manager. The destination offset for the response packet is the connectResponseOffset specified in the connection request packet. The format for the response packet is shown below.

If connectRequestStatus is not equal to CRS_SUCCESS, the connection manager may then free any resources associated with this connection. The connection manager shall unlock the connection registers, and return an error to the application.

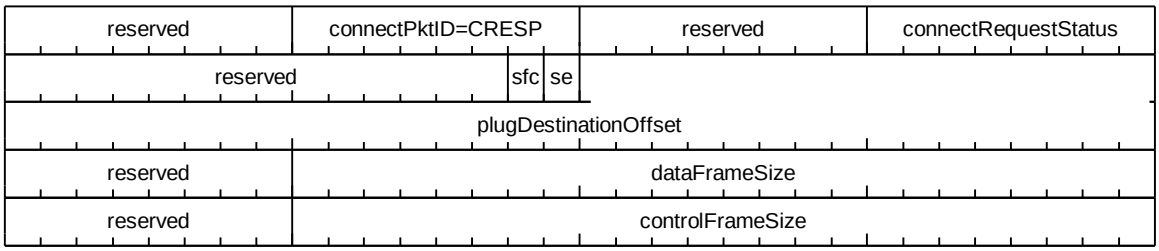

Figure 33 -- Connection response packet (CRESP)

- ***connectPktId*** identifies the connection packet as a response to a CREQ1 packet. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectRequestStatus*** is an indication of success or failure. A zero-value indicates success. See section 6.16.2.
- The ***sfc***-bit is one-valued if this node, acting as a producer, is capable of sending frames in small frame transfer mode.
- The ***se-***bit is an indication from this device of its tolerance for non-sequential writes. If se is one-valued, the connected node producer must write data sequentially to consumer segment buffers.
- ***plugDestinationOffset*** specifies the location in the device's own 1394 space for the newly created plug.
- ***dataFrameSize*** is the maximum number of bytes in a data frame sent from this device. If all 1's, the size of data frames is unknown at this time. If all 0's, there will be no data frames from this device. This field provides a hint for the connected device for sizing of the data path consumer segment buffers. See section 6.11.2.
- ***controlFrameSize*** is the maximum number of bytes in a control frame sent from this device. If all 1's, the size of control frames is unknown at this time. If all 0's, there will be no control frames sent from this device. This field provides a hint for the connected device for sizing of the control path consumer segment buffers. See section 6.11.2.

6.11.1.6 Connection manager sends connection request packet (CREQ1) to device2

The connection manager next sends a CREQ1 connection request packet to device2. The format for this packet was shown in Figure 32. In this case, ***node_ID*** and ***connectedNode_unique_ID*** pertain to device1. The destination offset is the connection register offset of device2 + 8 (bytes).

6.11.1.7 Device2 sends response packet (CRESP)

Device2 stores the information provided in the CREQ1 packet and formulates a connection response packet to be sent to the connection manager. The destination offset for the response packet is the connectResponseOffset specified in the connection request packet. The format for the response packet was shown previously in Figure 33.

6.11.1.8 Connection manager sends device2 information to device1 (CREQ2)

The connection manager now sends device2 plug information to device1. The format for this packet is shown below. The destination offset is the connection register of device1 + 8 (bytes).

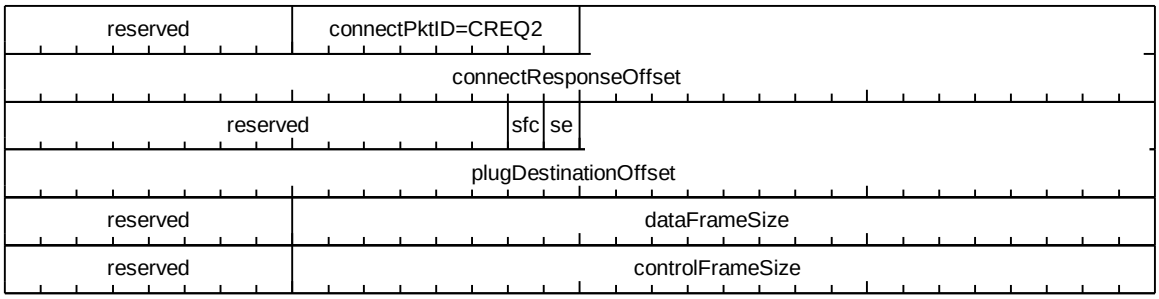

Figure 34 -- Connection request packet (CREQ2)

- ***connectPktId*** identifies the packet as a final connection request. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 1394 destination offset for the connection response packet.
- The ***sfc***-bit is one-valued if the connected node may utilize small frame transfer mode when sending small frames. The ***sfc***-bit is 0 if the connected node will never utilized small frame transfer mode.
- The ***se-***bit is one-valued if the connected node requires frame contents to be written sequentially.
- ***dataFrameSize*** is the number of bytes the connected producer will send in a data frame. See section 6.11.2.
- ***controlFrameSize*** is the total number of bytes the connected producer will send in a control frame. See section 6.11.2.

6.11.1.9 Device1 sends response packet (STATUS)

Device1 processes the connection request. It is recommended that device1 allocate segment buffer(s) for the data and control ports at this time. See section 6.11.2 below. Device1 then formulates a connection response packet. The destination offset is the connectResponseOffset specified in the connection request packet. The format of the response packet is shown below.

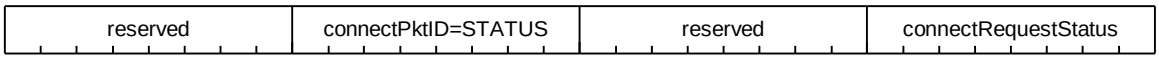

Figure 35 -- Connection response packet (STATUS)

- ***connectPktId*** identifies the connection packet as a connection response. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectRequestStatus*** is an indication of success or failure. A zero-value indicates success. See section 6.16.2.

If ***connectRequestStatus*** is not equal to CRS_SUCCESS, the connection manager shall unlock the connection registers and return an error to the application.

6.11.1.10 Connection manager sends device1 information to device2 (CREQ2)

The connection manager next sends a second connection request packet to device2, to convey information about device1. The format for the connection request packet is shown in Figure 34 (CREQ2). In this case, ***se***, ***plugDestinationOffset***, ***dataFrameSize***, ***controlFrameSize*** are attributes of the device1 plug. The destination offset for the CREQ2 packet is the connection register offset of device2 + 8 (bytes).

6.11.1.11 Device2 processes request packet, sends connection response

Device2 processes the connection request. Device2 formulates a connection response packet. The destination offset is the connectionResponseOffset specified in the request packet. The format of the response packet is shown in Figure 35 (STATUS).

If device2 unsuccessfully processes the request, device2 shall free any resources associated with this plug and return a ***connectRequestStatus*** not equal to CRS_SUCCESS. The connection manager shall send a connection request packet (connectPktID = FREE) to device1, unlock the connection registers, and return an error to the application.

6.11.1.12 Connection manager unlocks its connection register

The connection manager next unlocks its connection lock register with a simple write operation, clearing the 64-bit connection lock register.

6.11.1.13 Connection manager unlocks device1 connection register

If device1 is not the same device as the connection manager, unlocking of the device1 connection lock register is accomplished by using a 1394 compare and swap lock transaction, similar to the compare and swap lock request issued to lock the device1 connection register, as shown in Figure 27. For unlocking, the lock request ***arg_value*** shall be the 64-bit unique_ID of the connection manager that acquired the lock, and the lock request ***data_value*** shall be 0.

When the device1 connection register is unlocked, device1 may update plug registers on the connected node.

6.11.1.14 Connection manager unlocks device2 connection register

If device2 is not the same device as the connection manager, unlocking of the device2 connection lock register is accomplished by using a 1394 compare and swap lock transaction, similar to the compare and swap lock request issued to lock the device1 connection register, as shown in Figure 30. For unlocking, the lock request ***arg_value*** shall be the 64-bit unique_ID of the connection manager that acquired the lock, and the lock request ***data_value*** shall be 0.

When the device2 connection register is unlocked, device2 may update plug registers on the connected node.

6.11.2 Note on dataFrameSize, controlFrameSize and sizing of consumer buffers

The sizing of consumer buffers is highly implementation dependent. However, the table below illustrates a recommended strategy.

Producer frame size (dataFrameSize or controlFrameSize)	Recommended consumer large frame segment buffer size	Recommended consumer small frame Segment buffer size
frame size == 0	0 – no buffer needed	0 – no buffer needed
1 <= frame size <= MAX_BUF_SIZE	frame size, rounded up to nearest quadlet size to MAX_BUF_SIZE	If sfc-bit == 1, the recommended size is >= 2 Kbytes. If sfc-bit == 0, no buffer needed.
frame size > MAX_BUF_SIZE	MAX_BUF_SIZE	
frame size == FF $FFFF_{16}$ (unknown)	MAX_BUF_SIZE	

Table 5 – Consumer segment buffer size and dataFrameSize, controlFrameSize

MAX_BUF_SIZE is implementation dependent.

Note that the frame size specification is only indicating the maximum possible frame size. Producers may specify a large frame size and still send small frames. For this reason, consumers should allocate space for receiving small frames, so long as the producer has indicated that they are capable of sending small frames.

6.12 Connection deactivation

Generally, a device sets all IICP plug producer and consumer state machines to the deactivated state after a 1394-bus reset. The reason is to prevent plug communications to an incorrect node_ID. Node_ID's may change after a bus reset.

No updates to the public plug 1394 space are allowed, and no frame transfer activity is allowed until the connection has been reactivated. If deactivated, and a plug access occurs, and the implementation allows the specification of a response code, the response code shall be resp_conflict_err.

After a bus reset, all nodes with an IICP connection shall start a reactivation timer. Any further bus resets will cause the timer to be restarted. The connection manager that created the IICP connection shall issue a reactivation request as soon as possible after a bus reset. If a reactivation request is not received within time = REACT_TIMEOUT (see section 6.17 for value), a node experiencing resource shortages that can be remedied by freeing plug resources may discard and release plug resources associated with the timed out connection.

6.13 Connection reactivation

When a 1394 connection plug is deactivated, a reactivation sequence is used to reactivate the connection. It is recommended that connection managers reactivate connections before instantiating any new connections.

The connection manager that issued the connection sequence to create the plug is responsible for issuing a reactivation sequence for the plug.

For each of the plugs a connection manager created, the connection manager shall go through a reactivation sequence. The general flow is shown below. See section 6.18 for the connection manager state machine and section 6.19 for the connection client state machine for complete details.

Step	Device1		Connection manager		Device2
1			Lock connection register.		
2		←	Lock device1 connection register.		
3			Lock device2 connection register.	→	
4		←	Send reactivation request packet to device1 (REACT).		
5	Process request. Send connection response packet (STATUS).	→			
6			Send reactivation request packet to device2 (REACT).	→	
7				←	Process request. Send connection response packet (STATUS).
8			Unlock connection register.		
9		←	Unlock device1 connection register.		
10	May now communicate to Device2.		Unlock device2 connection register.	→	
11					May now communicate to Device1.

Figure 36 - Reactivation sequence

6.13.1 Reactivation sequence

The details for each of the steps in the table are given below.

6.13.1.1 Connection manager locks its own connection register

The connection manager shall first lock its own connection register as described in section 6.11.1.1.

6.13.1.2 Connection manager locks the connection register of device1

The connection manager shall next lock the device1 connection register. This was discussed in section 6.11.1.2.

6.13.1.3 Connection manager locks the connection register of device2

The connection manager shall next lock the device2 connection register. This was discussed in section 6.11.1.3.

6.13.1.4 Connection manager sends reactivation request (REACT) to device1

The connection manager sends a reactivation request packet to the device1 connection register. The format of the packet is shown below. The destination offset is the connection register of device1 + 8 (bytes).

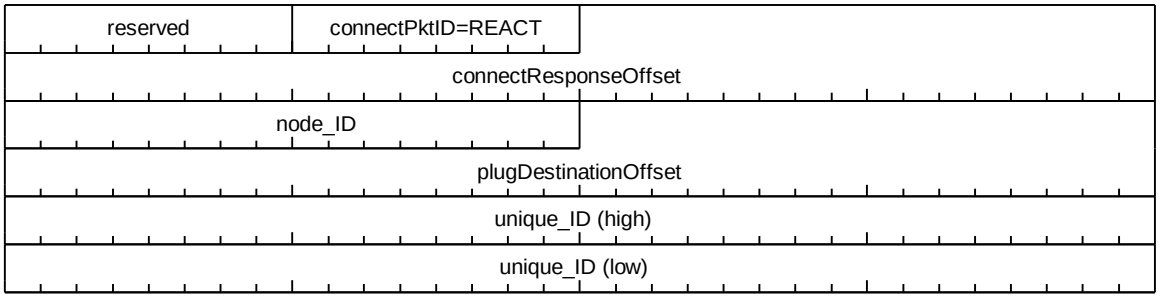

Figure 37 -- Reactivation request packet (REACT)

- ***connectPktId*** identifies the connection packet as a reactivation request. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 48-bit destination offset for the connection response packet.
- ***plugDestinationOffset*** specifies the destination offset of the deactivated plug on device1.
- ***node_ID*** is the possibly new 16-bit node_ID of device2.
- ***unique_ID*** is the 64-bit unique_ID for the connection manager issuing this request.

6.13.1.5 Device1 processes reactivation request and sends response packet (STATUS)

Device1 determines if it owns a plug with plugDestinationOffset that was created by the connection manager with the specified unique_ID. If it does, and if the connection has in fact been deactivated, device1 will reactivate the connection, updating the plug information with the new node_ID for device2. The producer and consumer state machines are restored to their previous state prior to the bus reset. However, no plug activity is permitted until the connection register is unlocked.

Device1 shall return a response packet. The format of the response packet is shown in Figure 35 (STATUS). The destination offset is the connectResponseOffset specified in the connection request packet.

If the reactivation is not successful, the connection manager shall unlock the connection registers. Reactivation retries are allowed within the REACT_TIMEOUT limit.

6.13.1.6 Connection manager sends reactivation request (REACT) to device2

The connection manager sends a reactivation request packet to the device2 connection register. The format of the packet is shown in Figure 37 (REACT). The destination offset is the connection register of device2 + 8 (bytes). The node_ID is the possibly new node_ID for device1.

6.13.1.7 Device2 processes reactivation request and sends response packet (STATUS)

Device2 determines if it owns a plug with plugDestinationOffset that was created by the connection manager with the specified unique_ID. If it does, and if the connection has in fact been deactivated, device2 will reactivate the connection, updating the plug information with the new node_ID for device1. The producer and consumer state machines are restored to their previous state.

Device2 shall return a response packet. The packet format is shown in Figure 35 (STATUS). The destination offset is the connectResponseOffset specified in the request packet.

If connectRequestStatus indicates the reactivation is not successful, the connection manager shall send a FREE request to the device1 plug that was reactivated.

6.13.1.8 Connection manager unlocks its connection register

The connection manager next unlocks its own lock register as described in section 6.11.1.12.

6.13.1.9 Connection manager unlocks device1 connection register

The connection manager next unlocks the device1 connection lock register as described in section 6.11.1.13.

6.13.1.10 Connection manager unlocks device2 connection register

The connection manager next unlocks the device2 connection lock register as described in section 6.11.1.14.

6.14 Disconnecting IICP connections

An IICP connection may be disconnected when no longer required. For each plug to be disconnected, the connection manager goes through the general flow shown in the table below. If device1 or device2 no longer exist, steps involving the non-existent device are skipped. See section 6.18 for the connection manager state machine and section 6.19 for the connection client state machine for complete details.

Step	Device1		Connection manager		Device2
1			Lock connection register		
2		←	Lock device1 connection register.		
3			Lock device2 connection register.	→	
4		←	Send connection request packet (STOP) to device1.		
5	Process request. Outgoing frame transfer activity should stop. Send response packet.	→			
6			Send connection request packet (STOP) to device2.	→	
7				←	Process request. Outgoing frame transfer activity should stop. Send response packet.
8		←	Send connection request packet (FREE) to device1.		
9	Free plug resources. Send response packet.	→			
10			Send connection request packet (FREE) to device2.	→	
11				←	Free plug resources Send response packet
12			Unlock connection register		
13		←	Unlock device1 connection register.		
14			Unlock device2 connection register.	→	

Figure 38 -- Disconnect sequence

The details for each step are shown below.

6.14.1 Disconnection sequence

6.14.1.1 Connection manager locks its own connection register

The connection manager shall first lock its own connection register. This is done as described in section 6.11.1.1.

6.14.1.2 Connection manager locks device1 connection register.

The connection manager shall next lock the device1 connection register. This was described in section 6.11.1.2.

6.14.1.3 Connection manager locks device2 connection register

The connection manager shall next lock the device2 connection register. This was described in section 6.11.1.3.

6.14.1.4 Connection manager sends STOP request packet to device1

The connection manager next sends a STOP request packet to device1. The format of the STOP request packet is shown below. The destination offset is the connection register of device1 + 8 (bytes).

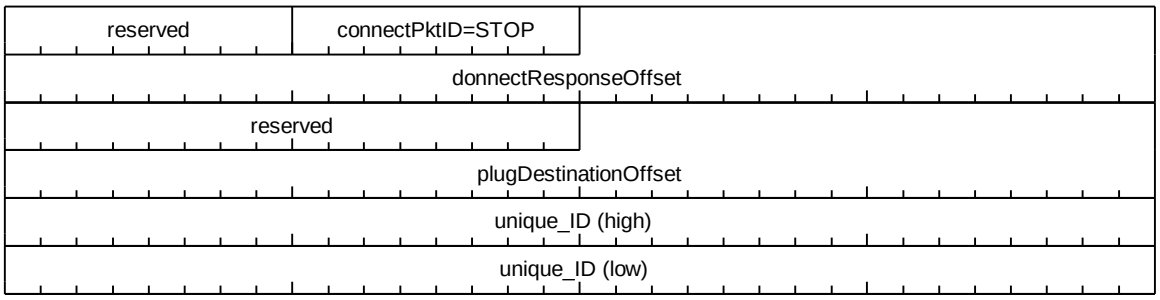

Figure 39 -- STOP request packet

- ***connectPktId*** identifies the connection packet as a STOP request. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 1394 destination offset for the connection response packet, set to the connection manager lock register offset + 8(bytes).
- ***plugDestinationOffset*** is the 1394 destination offset for the plug to be stopped.
- ***unique_ID*** is the 64-bit unique_ID for the connection manager.

6.14.1.5 Device1 processes STOP request and sends response packet

When device1 sees the STOP request packet, device1 should stop frame transfer activity. If any part of a frame has been transferred, the remaining part of the frame shall not be sent. No SmallFrameConsumer or LargeFrameConsumer update is done. After device1 has processed the STOP request and has stopped all producer activity on the plug, device1 sends a response packet as shown in Figure 35 (STATUS). The 1394 destination offset is the connectResponseOffset identified in the request packet.

6.14.1.6 Connection manager sends STOP packet to device2

The connection manager next sends a STOP request packet to device2. The format of the disconnection packet was shown in Figure 39. The destination offset used in sending the request packet is the connection register of device2 + 8 (bytes).

6.14.1.7 Device2 processes STOP request and sends response packet

When device2 sees the STOP request packet, device2 shall immediately stop frame transfers. If any part of a frame has been transferred, the remaining part of the frame is not sent. No SmallFrameConsumer or LargeFrameConsumer update is done. After device2 has processed the STOP request and has stopped all producer activity on the plug, device2 sends a response packet as shown in Figure 35 (STATUS). The 1394 destination offset is the connectResponseOffset identified in the request packet.

6.14.1.8 Connection manager sends FREE request to device1

The connection manager sends a connection request to device1, telling device1 it may now free resources associated with the connection.

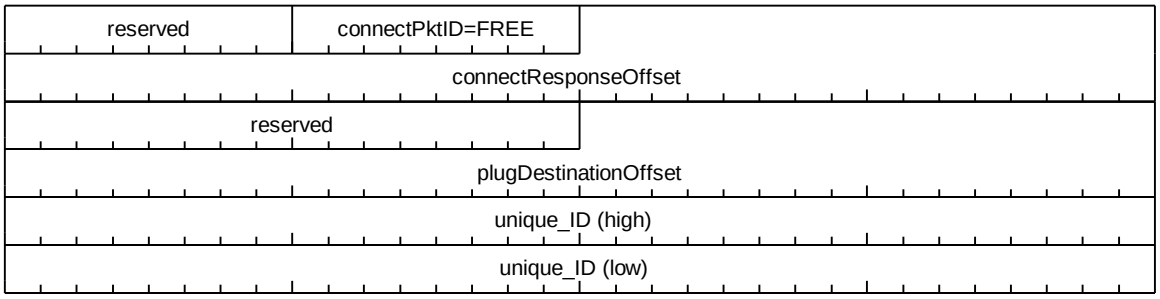

Figure 40 -- FREE request packet

- ***ConnectPktId*** identifies the connection packet as a FREE connection request. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 1394 destination offset for the connection response packet.
- ***plugDestinationOffset*** is the 1394 destination offset for the plug to be stopped.
- ***unique_ID*** is the 64-bit unique_ID for the connection manager.

6.14.1.9 Device1 sends response packet

Device1 frees the resources associated with the specified plug, and sends a response packet as shown in Figure 35 (STATUS). The 1394 destination offset is the connectResponseOffset identified in the request packet.

6.14.1.10 Connection manager sends FREE request to device2

The connection manager next sends a FREE connection request to device2, telling device2 it may now free resources associated with the connection.

6.14.1.11 Device2 sends response packet

Device2 frees the resources associated with the specified plug, and sends a response packet as shown in Figure 35 (STATUS). The 1394 destination offset is the connectResponseOffset identified in the request packet.

6.14.1.12 Unlock the connection register

The connection manager unlocks its own connection register. This is as described in section 6.14.1.12.

6.14.1.13 Unlock the device1 connection register

The connection manager unlocks the device1 connection register. This is as described in section 6.14.1.13.

6.14.1.14 Unlock the device2 connection register

The connection manager next unlocks the device2 connection register. This is as described in section 6.14.1.14.

6.15 Obtaining connection information

A device may wish to query other IICP devices to obtain information about the IICP connections that exist on a device. This may be useful after a connection manager reset, or for IICP debugging.

6.15.1 Connection information sequence

The device gathering information goes through the general flow shown in the table below. The device gathering information is referred to as the connection manager since it is responsible for locking the connection registers. The device may or may not have actually created the connection. The device giving the information is referred to as a connection client, since it is responding to connection register requests. See section 6.18 for the connection manager state machine and section 6.19 for the connection client state machine for complete details.

Step	Connection Manager (obtaining information)		Connection Client (providing information)
1	Lock connection manager connection register.		
2	Lock connection client connection register.	→	
3	Sends connect request packet (GETINFO) to connection client.	→	
4		←	Process request. Sends connect response packet (INFO).
5	Save information from INFO packet. Send GETPLUGINFO request to get information on plug #1 of N.	→	
6		←	Process request. Send response packet.
7	Repeat steps 5,6		
8	Unlock connection manager connection register		
9	Unlock connection client connection register	→	

Figure 41 -- Connection information sequence

The details for each step are shown below. Note that a connection manager may also lock the connection registers and then send a GETPLUGINFO for each plug of interest, then unlock the connection registers.

6.15.1.1 Connection manager locks its own connection register

The connection manager must lock its own connection register. This is discussed in section 6.11.1.1.

6.15.1.2 Connection manager locks the connection client lock register

The connection manager shall next lock the connection client connection register. This is discussed in section 6.11.1.2.

6.15.1.3 Connection manager sends GETINFO request to connection client

The connection manager next sends a connection request packet (GETINFO) to the connection client. The format of the request packet is shown below. The destination offset is the connection register of the connection client + 8 (bytes).

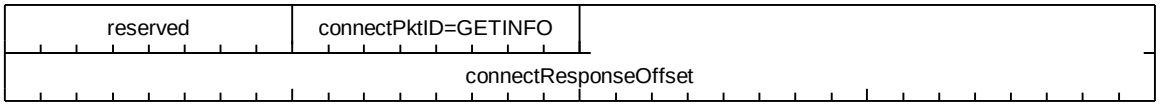

Figure 42 -- GETINFO request packet

- ***connectPktId*** identifies the connection packet as a request to get plug information. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 1394 destination offset for the connection response packet.

6.15.1.4 Connection client processes GETINFO and sends response packet

When the connection client sees the GETINFO request packet, the connection client generates a list of all current plugs. This list shall remain intact until the connection register is unlocked. The connection client formulates a response packet with the format shown below.

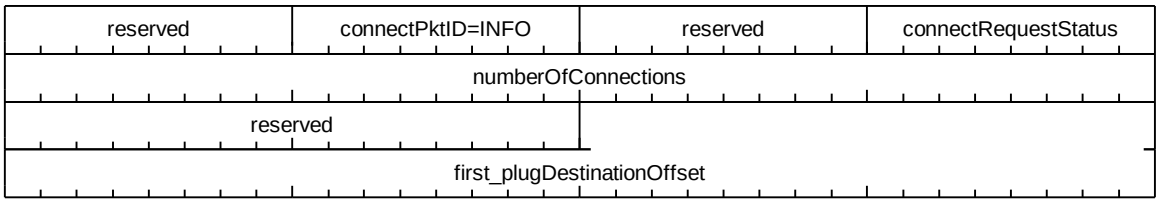

Figure 43 -- INFO response packet

- ***connectPktId*** identifies the connection packet as a connection response. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectRequestStatus*** is an indication of success or failure. A zero-value indicates success. See section 6.16.2.
- ***numberOfConnections*** specifies the number of IICP connections allocated on the connection client.
- ***first_plugDestinationOffset*** specifies a destination offset for the first plug in the list of plugs generated by the connection client.

If ***connectRequestStatus*** is not equal to CRS_SUCCESS, the connection manager shall unlock the connection registers and return an error to the application.

6.15.1.5 Connection manager sends GETPLUGINFO request to connection client

The connection manager may next obtain information about specific plugs on the connection client by sending a GETPLUGINFO request packet. The format of the request packet is shown below. The destination offset is the connection register of the connection client + 8 (bytes).

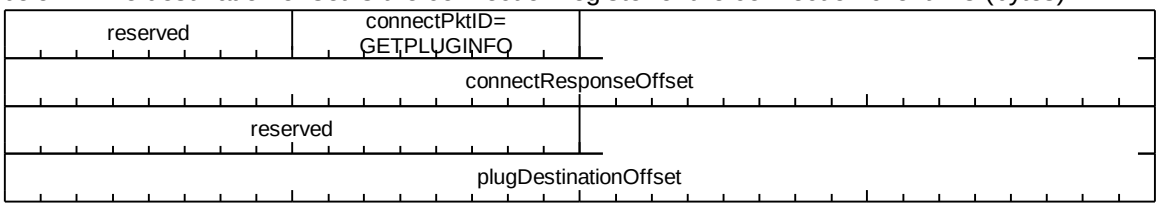

Figure 44 -- GETPLUGINFO request packet

- ***connectPktId*** identifies the connection packet as a request to get plug information. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectResponseOffset*** is the 1394 destination offset for the connection response packet.
- ***plugDestinationOffset*** specifies the plug the connection manager wishes to obtain information about.

6.15.1.6 Connection client processes GETPLUGINFO, sends PLUGINFO response packet

If the specified plug exists, the connection client sends a response packet, with information on the specified plug, as shown below. The 1394 destination offset is the connectResponseOffset identified in the request packet. If the specified plug does not exist, the connection client sends a response packet (STATUS) as shown in Figure 35, with connectRequestStatus = CRS_PARM.

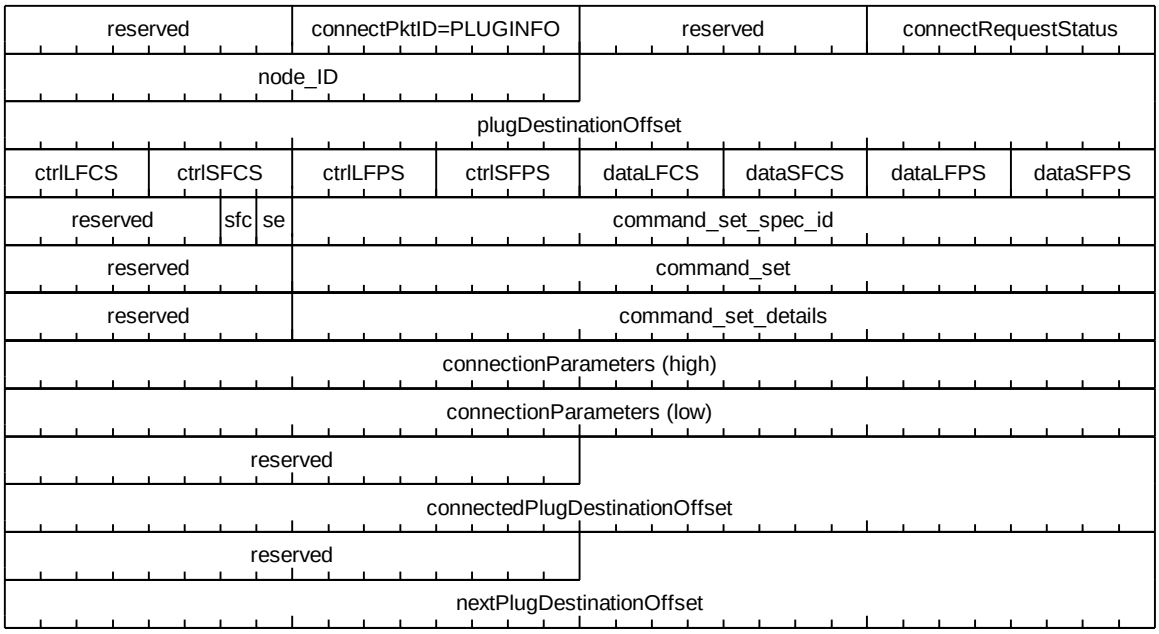

Figure 45 -- PLUGINFO response packet

- ***connectPktId*** identifies the connection packet as a connection response. See Table 6 -- ConnectPktId values in section 6.16.1.
- ***connectRequestStatus*** is an indication of success or failure. A zero-value indicates success. See section 6.16.2. If the GETPLUGINFO request was for a plug that does not exist, or there are no plugs on the connection client, the connectRequestStatus shall be set to CRS_UNKNOWN_PLUG.
- ***node_ID*** is the node_ID of the connected node.
- ***plugDestinationOffset*** specifies the destination offset of the plug.
- ***ctrlLFCS*** is the current state of the control port large frame consumer state machine. The consumer state is the decimal number following the ‘CLF’ in the state label. See section 6.20 for consumer state machine documentation.
- ***ctrlSFCS*** is the current state of the control port small frame consumer state machine. The consumer state is the decimal number following the ‘CSF’ in the state label. See section 6.20 for consumer state machine documentation.
- ***ctrlLFPS*** is the current state of the control port large frame producer state machine. The producer state is the decimal number following the ‘LFP’ in the state label. See section 6.21 for producer state machine documentation.
- ***ctrlSFPS*** is the current state of the control port small frame producer state machine. The producer state is the decimal number following the ‘SFP’ in the state label. See section 6.21 for producer state machine documentation.
- ***dataLFCS*** is the current state of the data port large frame consumer state machine. The consumer state is the decimal number following the ‘CLF’ in the state label. See section 6.20 for consumer state machine documentation.
- ***dataSFCS*** is the current state of the data port small frame consumer state machine. The consumer state is the decimal number following the ‘CSF’ in the state label. See section 6.20 for consumer state machine documentation.
- ***dataLFPS*** is the current state of the data port large frame producer state machine. The producer state is the decimal number following the ‘LFP’ in the state label. See section 6.21 for producer state machine documentation.
- ***dataSFPS*** is the current state of the data port small frame producer state machine. The producer state is the decimal number following the ‘SFP’ in the state label. See section 6.21 for producer state machine documentation.

- The ***sfc***-bit is one-valued if the connected node producer indicated it is capable of utilizing small frame transfer mode.
- The ***se***-bit is one-valued if the connected node requires sequential writes.
- ***command_set_spec_id*** indicates the original command_set_spec_id specified when the connection was established.
- ***command_set*** indicates the original command_set specified when the connection was established.
- ***command_set_details*** indicates the original command_set_details specified when the connection was established.
- ***connectionParameters*** indicates the original ConnectionParameters specified when the connection was established.
- ***connectedPlugDestinationOffset*** is the destination offset for the plug on the connected node.
- ***nextPlugDestinationOffset*** specifies the plugDestinationOffset for the next plug in the list on the connection client. If this is the last plug in the list generated when the GETINFO request packet was received, or if no GETINFO request preceded this GETPLUGINFO request, nextPlugDestinationOffset is set to FFFF FFFF $FFFF_{16}$.

The consumer and producer state values may be in the process of transition and therefore may be inaccurate. These values should be a best effort attempt to reflect the current state. If impossible to determine state, an implementation is allowed to return a value of FF_{16}.

If ***connectRequestStatus*** is not equal to CRS_SUCCESS, the connection manager shall unlock the connection registers and return an error to the application.

6.15.1.7 Repeat above 2 steps

The 2 steps above are repeated until the ***nextPlugDestinationOffset*** is 0 or a connectRequestStatus != CRS_SUCCESS is returned.

6.15.1.8 Connection manager unlocks its connection register

The connection manager unlocks its connection register. This is as described in section 6.14.1.12.

6.15.1.9 Connection manager unlocks connection client register

The connection manager next unlocks the connection client register. This is as described in section 6.14.1.14.

6.16 Summary of connection packet fields

6.16.1 ConnectPktId values

The table below enumerates the values for the 8-bit ConnectPktId.

ConnectPktId Macro	ConnectPktId Value (Decimal)	Definition
CREQ1	1	Connection request packet sent to begin a connection sequence.
CREQ2	2	Connection request packet sent that includes the connected node plug information.
REACT	3	Connection request packet issued to reactivate a connection.
STOP	4	Connection request packet issued to stop plug activity.
FREE	5	Free plug resources.
GETINFO	6	Get information about the total number of connections on a node.
GETPLUGINFO	7	Get information about a specific plug on a node.
CRESP	128	Connection response packet sent after processing a CREQ1.
STATUS	129	General response packet sent after receiving REACT, STOP, or FREE.
INFO	130	Response packet providing the total number of connections on a node.
PLUGINFO	131	Response packet providing information on a specific plug on a node.
	0,8-127,132-255	Reserved

Table 6 -- ConnectPktId values

6.16.2 connectRequestStatus values

connectRequestStatus is used in all connection response packets as an indication of success or failure. The table below summarizes connection response values.

connectRequestStatus Macro	connectRequestStatus Decimal Value	Definition
CRS_SUCCESS	0	Success
CRS_RSRC	1	Rejected- not enough resources
CRS_PARM	2	Rejected-illegal parameter in request
CRS_UNKNOWN_PLUG	3	Rejected-unknown plug
CRS_REG_NOT_LOCKED	4	Connection Register not locked, or not locked by the sender of the connection request.
CRS_NOT_IN_DEACTIVATED_STATE	5	Connection received a reactivation request and was not in the deactivated state.
CRS_NOT_STOPPED	6	Connection received a request to free plug resources but either the producer or consumer were not stopped yet.
CRS_BUS_RESET	7	A request failed due to a bus reset.
CRS_NO_DEV	8	A request fails because the device no longer is found on the bus.
CRS_CONNECT_REQ_TIMEOUT	9	Request failed because a connection request was sent and a connection response was not received in CONNECT_REQ_TIMEOUT.
	10-254	Reserved
CRS_FAIL	255	Failure, unspecified reason.

Table 7 -- connectRequestStatus values

6.17 Miscellaneous macro values

Time Macro	Decimal Value	Definition
CCLI_LOCK_TIMEOUT	10000	Number of milliseconds a connection client is allowed to remain locked. If this time expires, the connection client shall assume something anomalous occurred, and will unlock its own connection register.
REACT_TIMEOUT	10000	Number of milliseconds a node will wait after a bus reset for the connections to be reactivated. All connections not reactivated in this time may then be assumed to be stale, and the plug resources may be freed when necessary to complete new connection requests successfully.
CONNECT_REQ_TIMEOUT	1000	Number of milliseconds to wait after sending an acknowledged connection request packet before the sender of the packet will time out.

Table 8 -- Miscellaneous macro values

6.18 Connection manager state machine

The table below explains terms used in the following connection manager state machine.

Term	Explanation
BusReset	A 1394 bus reset occurrence.
CM_busReset()	Called by the connection manager to perform all necessary activities after a bus reset.
CMGR	Shorthand for the connection manager.
CMGR.lock	The connection lock register for the connection manager.
F	False = 0.
Indication(value)	Sets up a value to be returned to the caller that initiated the request. The full return value may be made up of more than one indication value, since some requests communicate to more than one device.
Lock(dev,arg_val, data_val)	Sends a 1394 lock request (compare and swap to a device) and waits for the lock response packet.
LockRegisters()	Locks the specified connection registers. See section 6.18.2.
QueueRequest(req)	Queue a connection request to the connection manager.
Send(dev,PktID)	Send a connection request packet to device=dev with connectPktID=PktID and waits for a connection response packet. Returns a connectRequestStatus value. If device fails to send a connection response packet and a connection request timeout occurs, the returned connectRequestStatus is set to CRS_CONNECT_REQ_TIMEOUT.
T	True = 1.
TestAndSet()	An atomic (non-interruptible) operation done on the connection manager lock register. If the lock register is 0-valued prior to this operation, the TestAndSet() will be successful and the value is set to CMGR unique_ID. If the lock register is not 0-valued prior to this operation, the TestAndSet() will fail. TestAndSet() may succeed if for some reason if the lock register value is already equal to the CMGR unique_ID.
UngetRequest()	Re-queues the current request so the request will be retried later.
UnlockRegisters()	Unlocks the specified connection registers. See section 6.18.3.

Table 9 -- Connection manager state machine terminology

6.18.1 Connection manager state machine: request startup

State	Transi-tion	Condition Action	New state
CM0: idle	TX0a	BusReset CM_busReset();	CM0
	TX0b	valid request received from higher layer 	CM1
	TX0c	Invalid request received from higher layer Indication(err)	CM0
CM1: LockReg LockRegisters()	TX1a	LockRegisters() fails due to previously locked condition Sleep(implementation-dependent time);	CM1
	TX1b	LockRegisters() fails due to bus reset UngetRequest();	CM0
	TX1c	LockRegisters() fails Indication(err);	CM0
	TX1d	LockRegisters() succeeds && request is to create a plug 	CM10
	TX1e	LockRegisters() succeeds && request is to reactivate a plug 	CM20
	TX1f	LockRegisters() succeeds && request is to stop a plug 	CM30
	TX1g	LockRegisters() succeeds && request is to get plug information. 	CM40
	TX1h	LockRegisters() succeeds && request is to get information on a specific plug only. 	CM50

Figure 46 -- Connection manager state machine: request startup

State CM0. The connection manager is idle.
Transition TX0a. A bus reset occurs.
Transition TX0b. A valid request is received from a higher protocol layer or from an application.
Transition TX0c. An invalid request is received from a higher protocol layer or from an application. An error indication is returned.

State CM1. The connection manager calls LockRegisters() to lock the connection registers of all devices to be involved in the connection request.
Transition TX1a. LockRegisters() fails due to a previously locked condition. The connection manager shall retry after an implementation dependent time.
Transition TX1b. LockRegisters() fails due to a bus reset. UngetRequest() re-queues the original request.
Transition TX1c. LockRegisters() fails due to some other condition. An error indication is returned.
Transition TX1d. LockRegisters() succeeds and the request is to create a plug.
Transition TX1e. LockRegisters() succeeds and the request is to reactivate a plug.
Transition TX1f. LockRegisters() succeeds and the request is to stop plug activity.
Transition TX1g. LockRegisters() succeeds and the request is to get plug information.
Transition TX1h. LockRegisters() succeeds and the request is to get information for a specific plug.

6.18.2 Connection manager state machine: LockRegisters(device1,device2)

State	Transi-tion	Condition Action	New state
L0: TestAndSet() on CMGR.lock	TX0a	BusReset CMGR.lock = 0; Indication(not locked due to bus reset); CM_busReset();	Return to caller
	TX0b	TestAndSet() fails Indication(not locked due to previously locked condition);	Return to caller
	TX0c	device1 != CMGR 	L1
	TX0d	Lock2 == 1 	L2
	TX0e	 Indication(lock success);	Return to caller
L1: Lock(device1, arg_value=0, data_value= CMGR uid);	TX1a	BusReset CMGR.lock = 0; Indication(not locked due to bus reset); CM_busReset();	Return to caller
	TX1b	Lock response indicates successful lock && Lock2 	L2
	TX1c	Lock response indicates device1 not successfully locked CMGR.lock = 0; Indication(not locked due to locked condition);	Return to caller
	TX1d	Lock response indicates successful lock && !Lock2 Indication(lock success);	Return to caller
L2: Lock (device2, arg_value=0, data_value= CMGR uid);	TX2a	BusReset CMGR.lock = 0; Indication(not locked due to bus reset); CM_busReset();	Return to caller
	TX2b	Lock response indicates successful lock Indication(lock success);	Return to caller
	TX2c	Lock response indicates device2 not successfully locked CMGR.lock = 0; UnlockRegisters(Unlock2=F); Indication(not locked due to locked condition);	Return to caller

Figure 47 -- Connection manager state machine: LockRegisters()

The call to LockRegisters() specifies a device (device1) or devices (device1, device2) to be locked, along with a variable Lock2 that determines if a second device, device2, is to be locked. The connection manager may be the same as device1.

State L0. Upon entry, the connection manager attempts an atomic test-and-set operation on the connection manager's connection lock register, abbreviated as CMGR.lock.
Transition TX0a. A bus reset occurs. The connection manager lock register is cleared and an indication is returned indicating the lock failed due to a bus reset. CM_busReset() is called.
Transition TX0b. The connection manager's lock register is already locked by another node. An indication is returned indicating the lock failed due to a locked condition.
Transition TX0c. All preceding conditions tested false, and device1 is not the connection manager.
Transition TX0d. All preceding conditions tested false, and there is a second device is to be locked.
Transition TX0e. All preceding conditions tested false, and no additional devices are to be locked. A successful lock indication is returned.

State L1. Upon entry, a lock request is sent to device1 in an attempt to lock the device1 connection lock register. The connection manager then waits for a lock response.
Transition TX1a. A bus reset occurs. The connection manager's lock register is cleared. An "unlocked due to bus reset" indication is returned. CM_busReset() is called.
Transition TX1b. Device1 returns a lock response indicating the lock request succeeded and there is a second device to be locked.
Transition TX1c. Device1 returns a lock response indicating the lock request failed. The connection manager's lock register is cleared. An "unlocked due to previously locked condition" indication is returned.
Transition TX1d. Device1 returns a lock response indicating the lock request succeeded and there is not a second device to be locked. A successfully locked indication is returned.

State L2. Upon entry, a lock request is sent to device2 in an attempt to lock the device2 connection lock register. The connection manager then waits for a lock response.
Transition TX2a. A bus reset occurs. The connection manager's lock register is cleared. An "unlocked due to bus reset" indication is returned. CM_busReset() is called.
Transition TX2b. Device2 returns a lock response indicating the lock request succeeded. A successful lock indication is returned.
Transition TX2c. Device2 returns a lock response indicating the lock request failed due to a previously locked condition. UnlockRegisters(Unlock2=F) is called to unlock all locked connection registers. An indication of "unlocked due to previously locked condition" is returned.

6.18.3 Connection manager state machine: UnlockRegisters()

State	Transi-tion	Condition Action	New state
UL0: UnlockResult = 0; CMGR.lock = 0;	TX0a	BusReset before CMGR.lock = 0 Indication(UnlockResult |= 1); CM_busReset();	Return to caller
	TX0b	BusReset after CMGR.lock = 0 Indication(UnlockResult |= 2); CM_busReset();	Return to caller
	TX0c	(CMGR == device1) && !Unlock2 Indication(UnlockResult);	Return to caller
	TX0d	(CMGR == device1) && Unlock2	UL2
	TX0e	CMGR != device1	UL1
UL1: Lock (device1, arg_value = CMGR uid, data_value=0);	TX1a	BusReset before lock request sent Indication(UnlockResult |= 0x2); CM_busReset();	Return to caller
	TX1b	BusReset after lock request sent && lock response not received Indication(UnlockResult |= 0x3); CM_busReset();	Return to caller
	TX1c	BusReset after lock response received Indication(UnlockResult |= 0x4); CM_busReset();	Return to caller
	TX1d	Lock response indicates successful unlock && Unlock2	UL2
	TX1e	Lock response indicates successful unlock && !Unlock2 Indication(UnlockResult);	Return to caller
	TX1f	Lock response indicates unlock failure && Unlock2 UnlockResult |= 0x10	UL2
	TX1g	Lock response indicates unlock failure && !Unlock2 Indication(UnlockResult |= 0x10);	Return to caller
UL2: Lock (device2, arg_value = CMGR uid, data_value=0);	TX2a	BusReset before lock request sent Indication(UnlockResult |= 0x4); CM_busReset();	Return to caller
	TX2b	BusReset after lock request sent Indication(UnlockResult |= 0x5); CM_busReset();	Return to caller
	TX2c	BusReset after response received Indication(UnlockResult |= 0x6); CM_busReset();	Return to caller
	TX2d	Lock response indicates unlock failure Indication(UnlockResult |= 0x20) ;	Return to caller
	TX2e	Lock response indicates successful unlock Indication(UnlockResult);	Return to caller

Figure 48 -- Connection manager state machine: UnlockRegisters()

The call to UnlockRegisters() specifies a device (device1) or devices (device1,device2) to be unlocked. Variable Unlock2 determines if a second device, device2, is to be unlocked. The connection manager may be the same as device1. The caller of UnlockRegisters(), in some cases, needs to know the success or failure of the operation, and may need to know if a bus reset occurred. For this reason, variable UnlockResult contains information on if and when a bus reset occurred. Additional information indicates the success or failure of unlocking CMGR, device1, and device2. A suggested definition for the UnlockResult bits is shown below.

UnlockResult Bit	Meaning
0,1,2	If non-zero, indicates point in the unlock procedure that a bus reset occurred. 0 – no bus reset 1 – bus reset before CMGR unlocked 2 – bus reset after CMGR unlocked and before unlock request sent to device1. 3 – bus reset after unlock request sent to device1 and before lock response. 4 – bus reset after unlock response from device1 and before unlock request sent to device2. 5 – bus reset after unlock request sent to device2 and before lock response. 6 – bus reset after unlock response from device2.
3	If non-zero, indicates error in unlocking the connection manager. This should never occur, but is provided for completeness.
4	If non-zero, indicates an error in unlocking device1.
5	If non-zero, indicates an error in unlocking device2.

Table 10 -- Suggested UnlockResult bit definitions

State UL0. Upon entry, UnlockResult is initialized to 0 and the connection manager lock register is cleared.
Transition TX0a. A bus reset occurs before the connection manager's lock register is unlocked. UnlockResult is modified to indicate this condition and is returned to the caller. CM_busReset() is called.
Transition TX0b. A bus reset occurs after the connection manager's lock register has been unlocked. UnlockResult is modified to indicate this condition and is returned to the caller. CM_busReset() is called.
Transition TX0c. The connection manager is the same as device1, and Unlock2 is false. There are no more devices to unlock. A successfully unlocked indication is returned to the caller.
Transition TX0d. The connection manager is the same as device1, and Unlock2 is true.
Transition TX0e. The connection manager is not the same as device1.

State UL1. Upon entry, a compare and swap lock request is sent to device1 in an attempt to unlock the device1 lock register. The connection manager waits for a lock response.
Transition TX1a. A bus reset occurs before the unlock request is sent to device1. UnlockResult is modified to indicate this event. UnlockResult is returned to the caller. CM_busReset() is called.
Transition TX1b. A bus reset occurs after the unlock request is sent to device1 and before the response is received. UnlockResult is modified to indicate this event. UnlockResult is returned to the caller. CM_busReset() is called.
Transition TX1c. A bus reset occurs after the unlock response returns. UnlockResult is modified to indicate this event. UnlockResult is returned to the caller. CM_busReset() is called.
Transition TX1d. Device1 successfully unlocked and Unlock2 indicates device2 needs to be unlocked.
Transition TX1e. Device1 successfully unlocked and there are no more devices to unlock. UnlockResult is returned to the caller.
Transition TX1f. Device1 unlock fails. UnlockResult is modified to indicate this condition. Device2 remains to be unlocked.
Transition TX1g. Device1 unlock fails. There are no more devices to be unlocked. UnlockResult is modified to indicate this condition. UnlockResult is returned to the caller.

State UL2. Upon entry, a compare-swap lock request is sent to device2 in an attempt to unlock the device2 lock register. The connection manager waits for a lock response packet.
Transition TX2a. A bus reset occurs before the unlock request is sent to device2. UnlockResult is modified to indicate this and is returned to the caller. CM_busReset() is called.
Transition TX2b. A bus reset occurs after the unlock request has been sent but before a response is received. UnlockResult is modified to indicate this and is returned to the caller. CM_busReset() is called.
Transition TX2c. A bus reset occurs after the response is received. UnlockResult is modified to indicate this and is returned to the caller. CM_busReset() is called.

Transition TX2d. Device2 unlock fails. UnlockResult is modified to indicate this and is returned to the caller.
Transition TX2e. Device2 unlock succeeds. UnlockResult is returned to the caller.

6.18.4 Connection manager state machine: creating a plug

State	Transition	Condition / Action	New state
CM10: Err = Send(device1, CREQ1);	TX10a	BusReset UngetRequest(); CM_busReset();	CM0
	TX10b	Err == CRS_SUCCESS 	CM11
	TX10c	Err != CRS_SUCCESS UnlockRegisters(Unlock2=T); Indication(Err);	CM0
CM11: Err = Send(device2, CREQ1);	TX11a	BusReset UngetRequest();CM_busReset();	CM0
	TX11b	Err == CRS_SUCCESS 	CM12
	TX11c	Err != CRS_SUCCESS UnlockRegisters(Unlock2=T); Indication(Err);	CM0
CM12: Err = Send(device1, CREQ2);	TX12a	BusReset UngetRequest();CM_busReset();	CM0
	TX12b	Err == CRS_SUCCESS 	CM13
	TX12c	Err != CRS_SUCCESS UnlockRegisters(Unlock2=T); Indication(Err);	CM0
CM13: Err = Send(device2, CREQ2);	TX13a	BusReset before response received UngetRequest(); CM_busReset();	CM0
	TX13b	Err == CRS_SUCCESS 	CM14
	TX13c	Err != CRS_SUCCESS Err2 == Err	CM15
CM14: UnlockRegisters(Unlock2=T);	TX14a	UnlockRegisters() fails due to bus reset before device1 unlocked. UngetRequest();CM_busReset();	CM0
	TX14b	UnlockRegisters() fails due to bus reset before device2 unlocked. QueueRequest(device1 STOP,FREE); UngetRequest(); CM_busReset();	CM0
	TX14c	UnockRegisters() returns success or fails after device2 unlocked. Indication(success); QueueRequest(reactivate plug);	CM0
CM15: Err = Send(device1, FREE);	TX15a	BusReset Indication(Err); CM_busReset();	CM0
	TX15b	Response received Indication(Err2); UnlockRegisters(Unlock2=T);	CM0

Figure 49 -- Connection manager state machine: creating a plug

State CM10. This state is entered after the connection registers have been locked and the connection request involves creating a new plug. Upon entry a CREQ1 connection request packet is sent to device1.
Transition TX10a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX10b. Device1 returns a CRESP packet with connectRequestStatus == CRS_SUCCESS.
Transition TX10c. Device1 returns a packet with connectRequestStatus != CRS_SUCCESS. UnlockRegisters() is called to unlock any locked connection registers. An error indication is returned.

State CM11. Upon entry, a CREQ1 packet is sent to device2.

Transition TX11a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX11b. Device2 returns a CRESP packet with connectRequestStatus == CRS_SUCCESS.
Transition TX11c. Device2 returns a packet with connectRequestStatus != CRS_SUCCESS. UnlockRegisters() is called to unlock any locked connection registers. An error indication is returned.

State CM12. Upon entry, a CREQ2 packet is sent to device1.
Transition TX12a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX12b. Device1 returns a STATUS packet with connectRequestStatus == CRS_SUCCESS.
Transition TX12c. Device1 returns a STATUS packet with connectRequestStatus != CRS_SUCCESS. UnlockRegisters() is called to unlock any locked connection registers. An error indication is returned.

State CM13. Upon entry, a CREQ2 packet is sent to device2.
Transition TX13a. A bus reset occurs before a response is received. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX13b. Device2 returns a STATUS packet with connectRequestStatus == CRS_SUCCESS.
Transition TX13c. Device2 returns a STATUS packet with unexpected connectRequestStatus. Err2 is set to the connectRequestStatus.

State CM14. Upon entry, UnlockRegisters(Unlock2=T) is called to unlock the connection registers.
Transition TX14a. UnlockRegisters() fails due to a bus reset before device1 is unlocked. UngetRequest() is called to re-queue the original request.
Transition TX14b. UnlockRegisters() fails due to bus reset after device1 is unlocked but before device2 is unlocked. A request to STOP and FREE the newly created plug on device1 is queued. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX14c. UnlockRegisters() returns success or returns failure, but the failure occurs after device2 unlocked. The connection remains valid. QueueRequest() is called to queue a reactivation request for the newly created plug.

State CM15. Upon entry, a FREE request is sent to device1.
Transition TX15a. A bus reset occurs. The error indication from device2 is returned. CM_busReset() is called.
Transition TX15b. A response is received. UnlockRegisters() is called. The error indication from device2 is returned.

.

6.18.5 Connection manager state machine: reactivating a connection

State	Transi-tion	Condition Action	New state
CM20 Err = Send(device1, REACT);	TX20a	BusReset CM_busReset();	CM0
	TX20b	(Err == CRS_SUCCESS) || (Err == CRS_NOT_IN_DEACTIVATED_STATE) 	CM21
	TX20c	 UnlockRegisters(Unlock2=T); Indication(Err);	CM0
CM21 Err = Send(device2, REACT);	TX21a	BusReset CM_busReset();	CM0
	TX21b	(Err == CRS_SUCCESS) || (Err == CRS_NOT_IN_DEACTIVATED_STATE) 	CM23
	TX21c	 Err2 = Err;	CM22
CM22 Err = Send(device1, FREE);	TX22a	BusReset Indication(Err2); CM_busReset();	CM0
	TX22b	Response received UnlockRegisters(Unlock2=T); Indication(Err2);	CM0
CM23 UnlockRegisters(Unlock2=T);	TX23a	UnlockRegisters() completes Indication(Err);	CM0

Figure 50 -- Connection manager state machine: reactivating a plug

State CM20. This state is entered after the connection registers have been locked and the connection request involves reactivating a plug. Upon entry, a REACT connection request packet is sent to device1.
Transition TX20a. A bus reset occurs. CM_busReset() will reschedule a reactivation request.
Transition TX20b. Device1 returns a STATUS response packet with connectRequestStatus == CRS_SUCCESS or CRS_NOT_IN_DEACTIVATED_STATE.
Transition TX20c. Device1 returns a response packet with an unexpected connectRequestStatus. UnlockRegisters() is called to unlock any locked connection registers. An error indication is returned.

State CM21. Upon entry, a REACT connection request packet is sent to device2.
Transition TX21a. A bus reset occurs. CM_busReset() is called and will reschedule a reactivation request.
Transition TX21b. Device2 returns a response packet with connectRequestStatus == CRS_SUCCESS or CRS_NOT_IN_DEACTIVATED_STATE.
Transition TX21c. Device2 returns a response packet with an unexpected connectRequestStatus. Variable Err2 is set to Err to be returned later.

State CM22. Upon entry, a FREE request packet is sent to device1 to free the plug.
Transition TX22a. A bus reset occurs. Error indication Err2 is returned. CM_busReset() is called.
Transition TX22b. Device1 returns a response packet. UnlockRegisters() is called to unlock any locked connection registers. Error indication Err2 is returned.

State CM23. Upon entry, UnlockRegisters(Unlock2=T) is called to unlock connection registers.
Transition TX23a. UnlockRegisters() completes. An indication made up of Err is returned.

6.18.6 Connection manager state machine: stopping a connection

State	Transi-tion	Condition Action	New state
CM30 Err2=no error; Err1 = Send(device1, STOP);	TX30a	BusReset UngetRequest(); CM_busReset();	CM0
	TX30b	StopDevice2 	CM31
	TX30c	(Err1 != CRS_SUCCESS) UnlockRegisters(Unlock2=F); Indication(Err1,Err2);	CM0
	TX30d	 	CM32
CM31 Err2 = Send(device2, STOP);	TX31a	BusReset UngetRequest(); CM_busReset();	CM0
	TX31b	((Err2 == CRS_SUCCESS) || (Err2 == CRS_UNKOWN_PLUG) || (Err2 == CRS_NO_DEV)) && (Err1 == CRS_SUCCESS) 	CM32
	TX31c	(Err2 == CRS_SUCCESS) && ((Err1 == CRS_UNKNOWN_PLUG) || (Err1 == CRS_NO_DEV)) 	CM33
	TX31d	 UnlockRegisters(Unlock2=T); Indication(Err1,Err2);	CM0
CM32 Err1 = Send(device1, FREE);	TX32a	BusReset Indication(Err1, Err2); CM_busReset();	CM0
	TX32b	!StopDevice2 UnlockRegisters(Unlock2=F); Indication(Err1)	CM0
	TX32c	Err2 == CRS_SUCCESS 	CM33
	TX32d	 UnlockRegisters(Unlock2=F); Indication(Err1,Err2);	CM0
CM33 Err2 = Send(device2, FREE);	TX33a	BusReset Indication(Err1,Err2); CM_busReset();	CM0
	TX33b	 UnlockRegisters(Unlock2=T); Indication(Err1,Err2);	CM0

Figure 51 -- Connection manager state machine: stopping a connection

A stop plug request specifies a device (device1) or devices (device1, device2) to be stopped. Variable StopDevice2 determines if a second device is to be stopped. The conditions shall be evaluated in the order given.

State CM30. This state is entered after the connection registers have been locked and the request involves stopping plug activity. Upon entry, Err2 is initialized and a STOP connection request packet is sent to device1.
Transition TX30a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX30b. There is a second device to be stopped.
Transition TX30c. Device1 returns a connection response packet with connectRequestStatus != CRS_SUCCESS. The connection registers are unlocked and an error indication is returned.
Transition TX30d. All of the above conditions evaluate false.

State CM31. Upon entry, a STOP connection request packet is sent to device2.
Transition TX31a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX31b. Err2 == CRS_SUCCESS || Err2 == CRS_UNKNOWN_PLUG || Err2 == CRS_NO_DEV and device1 successfully processed its STOP request. It is safe to proceed by sending a FREE connection request packet to device1.

Transition TX31c. Err2 == CRS_SUCCESS and device1 was not successfully stopped, but it is safe to proceed and send FREE connection request packet to device2.
Transition TX31d. All of the above conditions evaluate false.

State CM32. Upon entry, a FREE request packet is sent to device1.
Transition TX32a. A bus reset occurs. An indication made up of Err1, Err2 is set up to be returned. CM_busReset() is called.
Transition TX32b. There is not a second device to be stopped.
Transition TX32c. Err2 == CRS_SUCCESS.
Transition TX32d. All of the above conditions evaluate false.

State CM33. Upon entry, a FREE connection request packet is sent to device2.
Transition TX33a. A bus reset occurs. CM_busReset() is called.
Transition TX33b. UnlockRegisters(Unlock2=T) is called. An indication made up of Err1, Err2 is returned.

6.18.7 Connection manager state machine: get plug information

State	Transi-tion	Condition Action	New state
CM40 Err = Send(device, GETINFO);	TX40a	BusReset UngetRequest(); CM_busReset();	CM0
	TX40b	Err != CRS_SUCCESS UnlockRegisters(Unlock2=F); Indication(Err);	CM0
	TX40c	NumberOfConnections == 0 UnlockRegisters(Unlock2=F); Indication(INFO);	CM0
	TX40d	NumberOfConnections > 0 Save information	CM41
CM41 Err = Send(device, GETPLUGINFO);	TX41a	BusReset UngetRequest(); CM_busReset();	CM0
	TX41b	Err != CRS_SUCCESS UnlockRegisters(Unlock2=F); Indication(Err);	CM0
	TX41c	More plug information to get Save information	CM41
	TX41d	no more plug information to get Save information; UnlockRegisters(Unlock2=F); Indication(success,INFO,accumulated plug information);	CM0

Figure 52 -- Connection manager state machine: get plug information

State CM40: Upon entry, the connection manager sends a GETINFO request packet to the device.
Transition TX40a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX40b. A response is received with connectRequestStatus != CRS_SUCCESS. The connection registers are unlocked. An error indication is returned.
Transition TX40c. The number of connections, or plugs, is 0. UnlockRegisters() is called. An indication made up of the INFO response packet is returned.
Transition TX40d. The number of connections, or plugs, is greater than 0.

State CM41. Upon entry, a GETPLUGINFO request packet is sent to the device.
Transition TX41a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX41b. A response packet is received with connectRequestStatus != CRS_SUCCESS. The connection registers are unlocked and an error is returned.
Transition TX41c. There is more information to get from the device.
Transition TX41d. There is no more information to get from the device. An indication made up of the INFO response packet and the accumulated PLUGINFO response packets is returned.

6.18.8 Connection manager state machine: get specific plug information

State	Transition	Condition / Action	New state
CM50 Err = Send(device, GETPLUGINFO);	TX50a	BusReset UngetRequest(); CM_busReset();	CM0
	TX50b	Err != CRS_SUCCESS UnlockRegisters(Unlock2=F); Indication(Err);	CM0
	TX50c	Err == CRS_SUCCESS UnlockRegisters(Unlock2=F); Indication(PLUGINFO);	CM0

Figure 53 -- Connection manager state machine: get specific plug information

State CM50: Upon entry, the connection manager sends a GETPLUGINFO request packet to the device.
Transition TX50a. A bus reset occurs. UngetRequest() re-queues the original request. CM_busReset() is called.
Transition TX50b. A response packet is received with connectRequestStatus != CRS_SUCCESS. The connection registers are unlocked and an error is returned.
Transition TX50c. A response packet is received with connectRequestStatus == CRS_SUCCESS. The connection registers are unlocked and the plug information is returned.

6.18.9 Connection manager state machine: CM_busReset()

This state machine documents the connection manager activities required after a bus reset.

State	Transition	Condition / Action	New state
CMBR0: De-queue all reactivation requests	TX0a	(CMGR.lock == CMGR unique_ID) || (CMGR.lock != 0 && CMGR.lock occurred prior to bus reset) CMGR.lock = 0;	CMBR1
	TX0b		CMBR1
CMBR1 Enumerate the 1394 bus.	TX1a	BusReset	CMBR0
	TX1b	Enumeration complete Queue reactivation requests;	CM0

Figure 54 -- Connection manager state machine: CM_busReset()

State CM_BR0: Upon entry, all reactivation requests that are queued are de-queued.
Transition TX0a. The connection manager lock register is currently locked by the connection manager or is locked by another node but the lock occurred prior to the bus reset event. The lock register is unlocked.
Transition TX0b. The above condition evaluates false.

State CM_BR1. Upon entry, the bus is enumerated. This involves the connection manager going through a process of discovery to find what IICP devices exist on the bus.
Transition TX1a. A bus reset occurs.
Transition TX1b. The enumeration is done. The connection manager queues the appropriate reactivation requests.

6.19 Connection client state machine

The table below explains terms used in the following connection client state machine.

Term	Explanation
BusReset	A 1394 bus reset occurrence.
CC_busReset()	Called by the connection client to perform all necessary activities after a bus reset. This will deactivate all plugs.
CCLI_UnlockEvent	Event indicating an action to be performed after the connection register is unlocked.
F	False = 0.
LockTimeout()	Returns true if the connection client timer has expired. The timer expires after CCLI_LOCK_TIMEOUT.
Send(pktID, status)	Sends a connection response packet with connectRequestPktID = pktID and connectRequestStatus = status. If a bus reset occurs, the send is aborted. If Send() fails, an error should be logged.
T	True = 1.
UnlockOK	Variable set true if connection client expects an unlock request.

Table 11 -- Connection client state machine terminology

6.19.1 Locking of connection register and waiting for request

State	Transi-tion	Condition Action	New state
CC0: unlocked UnlockOK=F;	TX0a	BusReset CC_busReset();	CC0
	TX0b	Valid lock request Send lock response; initialize LockTimeout() timer; ValidRequests = CREQ1, REACT, STOP,GETINFO, GETPLUGINFO; UnlockOK=T; CCLI_UnlockEvent = NULL;	CC1
	TX0c	Invalid lock request Send lock response	CC0
	TX0d	Connect request packet received Rcode = resp_type_error; Send(STATUS,CRS_NOT_LOCKED);	CC0
	TX0e	Connect response packet received Rcode = resp_type_error;	CC0
CC1: locked Wait for a connection request or for unlock request	TX1a	Bus reset Free any resources allocated since connection register was locked; Unlock connection lock register; CC_busReset();	CC0
	TX1b	Lock request == lock || invalid unlock request Send lock response	CC1
	TX1c	Lock request == valid unlock request && UnlockOK == T Unlock connection lock register; Send lock response; if (CCLI_UnlockEvent == Reactivation) Send ReactivationEvent to producer and consumer state machines. else if (CCLI_UnlockEvent == PlugCreation) Send CreationEvent to consumer state machines	CC0
	TX1d	Lock request == valid unlock request && UnlockOK == F Unlock connection lock register; Free any resources allocated since connection register was locked. Send lock response	CC0
	TX1e	LockTimeout() Free any resources allocated since connection register was locked; Unlock connection lock register;	CC0
	TX1f	Invalid request received Rcode = resp_type_error; Send(STATUS,CRS_PARM);	CC1
	TX1g	Connection response packet received. Rcode = resp_type_error;	CC1
	TX1h	Valid request received && request == CREQ1 UnlockOK=F;	CC10
	TX1k	Valid request received && request == CREQ2	CC20
	TX1m	Valid request received && request == REACT	CC30
	TX1n	Valid request received && request == STOP	CC40
	TX1p	Valid request received && request == FREE	CC50
	TX1q	Valid request received && request == GETINFO	CC60
	TX1r	Valid request received && request == GETPLUGINFO	CC70

Figure 55 -- Connection client state machine: locking and waiting for request

State CC0. The connection client lock register is unlocked.
Transition TX0a. A bus reset occurs. CC_busReset() is called.

Transition TX0b. A valid lock request is received. A lock response is sent. The LockTimeout() timer is initialized. A variable, ValidRequests, is initialized to the set of connection requests that will be treated as valid requests. Variable UnlockOK is set true. Variable CCLI_UnlockEvent is set to NULL, indicating there is not yet an event to be communicated when the connection register is unlocked.
Transition TX0c. An invalid lock request is received. An appropriate lock response is returned.
Transition TX0d. An unexpected connect request packet is received. A response code = resp_type_error should be returned. A response packet is sent, with connectPktID = STATUS and connectRequestStatus = CRS_NOT_LOCKED.
Transition TX0e. An unexpected connect response packet is received. A response code = resp_type_error should be returned.

State CC1. The connection client's lock register is locked. The connection client waits for a connection request or unlock.
Transition TX1a. A bus reset occurs. Any resources allocated in the handling of connection requests since the connection register was locked shall be freed. The connection client lock register is unlocked. CC_busReset() is called.
Transition TX1b. A lock request or invalid unlock request is received. An appropriate lock response is sent.
Transition TX1c. A valid, expected unlock request is received. The connection client lock register is unlocked. A lock response is sent. If CCLI_UnlockEvent is set, perform the appropriate communication to the consumer and producer state machines.
Transition TX1d. A valid, but unexpected, unlock request is received. The connection client lock register is unlocked. A lock response is sent. Any resources allocated in the handling of connection requests since the connection register was locked should be freed.
Transition TX1e. A LockTimeout() occurs. Any resources allocated in the handling of connection requests since the connection register was locked should be freed. The connection client lock register is unlocked.
Transition TX1f. An invalid connection request is received. A response code = resp_type_error should be returned. A response packet is sent, with connectPktID = STATUS and connectRequestStatus = CRS_PARM.
Transition TX1g. An invalid connection response packet is received. A response code = resp_type_error should be returned.
Transition TX1h – Transition TX1r. A valid connection request packet is received and the connection client state machine moves to the appropriate state to handle the request.

6.19.2 Connection client request == CREQ1

State	Transi-tion	Condition Action	New state
CC10: Process the CREQ1 request	TX10a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX10b	Processing of CREQ1 request successful. Send(CRESP,CRS_SUCCESS); ValidRequests = CREQ2	CC1
	TX10c	Processing of CREQ1 request fails Send(CRESP,appropriate status); ValidRequests = none;	CC1

Figure 56 -- Connection client state machine: CREQ1 processing

State CC10. The connection client processes the CREQ1 packet. This involves creating a new plug.
Transition TX10a. A bus reset occurs. Any resources allocated for the new plug are freed. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX10b. The plug is successfully created. A response packet is sent. The next valid request is CREQ2.
Transition TX10c. The plug creation failed or a higher layer is not yet ready to handle connection requests. A response packet is sent. The set of valid request packets is set to none.

6.19.3 Connection client request == CREQ2

State	Transition	Condition / Action	New state
CC20: Process the CREQ2 request	TX20a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX20b	Processing of CREQ2 successful. Send(STATUS,CRS_SUCCESS); ValidRequests = FREE; CCLI_UnlockEvent = CreationEvent;	CC1
	TX20c	Processing of CREQ2 fails Send(STATUS,appropriate status); ValidRequests = none;	CC1

Figure 57 -- Connection client state machine: CREQ2 processing

State CC20. The connection client processes the CREQ2 request.
Transition TX20a. A bus reset occurs. Any resources allocated for the new plug are freed. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX20b. The request is processed successfully. A response packet is sent. The only valid following request is FREE. Variable CCLI_UnlockEvent is set so when the connection register is unlocked the appropriate event is communicated to the consumer state machines.
Transition TX20c. The processing of the request failed. A response packet is sent. The set of valid request packets is set to none.

6.19.4 Connection client request == REACT

State	Transition	Condition / Action	New state
CC30: Check if plug is deactivated.	TX30a	BusReset Unlock connection register; CC_busReset();	CC0
	TX30b	Plug is deactivated Notify consumer and producer state machines of REACT received; Send(STATUS,CRS_SUCCESS); ValidRequests = none; CCLI_Event = Reactivation;	CC1
	TX30c	Plug not deactivated Send(STATUS,CRS_NOT_IN_DEACTIVATED_STATE); ValidRequests = none;	CC1

Figure 58 -- Connection client state machine: REACT processing

State CC30. The connection client checks if the plug is deactivated.
Transition TX30a. A bus reset occurs. The connection client's lock register is unlocked. CC_busReset();
Transition TX30b. The plug is deactivated. The connection client notifies the consumer and producer state machines. The consumer and producer state machines will restore their state to the state they were in prior to the bus reset. A response packet is sent. The set of valid requests is set to none. Variable CCLI_UnlockEvent is set so when the connection register is unlocked the appropriate event is communicated to the consumer and producer state machines.
Transition TX30c. The plug was not in the deactivated state. A response packet is sent. The set of valid request packets is set to none.

6.19.5 Connection client request == STOP

State	Transi-tion	Condition Action	New state
CC40: Process STOP request.	TX40a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX40b	Processing of STOP successful. Send(STATUS,CRS_SUCCESS); ValidRequests = FREE	CC1
	TX40c	Processing of STOP fails Send(STATUS,appropriate status); ValidRequests = none;	CC1

Figure 59 -- Connection client state machine: STOP processing

State CC40. The connection client processes the STOP request. The producer state machine is notified and it should stop sending frame content. The consumer state machine is notified of the STOP request and transitions to the appropriate state.
Transition TX40a. A bus reset occurs. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX40b. The processing of the request was successful. The device shall not send any more frame contents or perform any other plug activities. A response packet is sent. The set of valid requests is set to FREE.
Transition TX40c. The processing of the request failed. The device shall not send any more frame contents or perform any other plug activities. A response packet is sent. The set of valid request packets is set to none.

6.19.6 Connection client request == FREE

State	Transi-tion	Condition Action	New state
CC50: Notify consumer and producer state machines of FREE. Free plug resources.	TX50a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX50b	Processing of FREE successful. Send(STATUS,CRS_SUCCESS); ValidRequests = none;	CC1
	TX50c	Processing of FREE fails Send(STATUS,appropriate status); ValidRequests = none;	CC1

Figure 60 -- Connection client state machine: FREE processing

State CC50. Upon entry, the connection client notifies the consumer and producer state machines of the FREE packet having been received. Free the plug resources.
Transition TX50a. A bus reset occurs. All plug resources are freed. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX50b. The request was processed successfully. A response packet is sent. The set of valid requests is set to none.
Transition TX50c. The request was not processed successfully. A response packet is sent. The set of valid requests is set to none.

6.19.7 Connection client request == GETINFO

State	Transition	Condition / Action	New state
CC60: Generate information to be returned in the response packet.	TX60a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX60b	Processing of GETINFO successful. Send(INFO,CRS_SUCCESS); ValidRequests = GETPLUGINFO;	CC1
	TX60c	Processing of GETINFO fails Send(INFO,appropriate status); ValidRequests = none;	CC1

Figure 61 -- Connection client state machine: GETINFO processing

State CC60. Upon entry, the connection client generates a list of all known plugs. It also generates all the information to be returned in the response packet (connectPktID=INFO).
Transition TX60a. A bus reset occurs. All allocated resources for handling this request are freed. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX60b. The information has been gathered. A response packet is sent. The set of valid requests is set to GETPLUGINFO.
Transition TX60c. The processing of GETINFO failed. A response packet is sent. The set of valid requests is set to none.

6.19.8 Connection client request == GETPLUGINFO

State	Transition	Condition / Action	New state
CC70: Generate information about the specified plug and a handle for the next plug in the list.	TX70a	BusReset Free any resources allocated; Unlock connection register; CC_busReset();	CC0
	TX70b	Processing of GETPLUGINFO successful. Send(INFO,CRS_SUCCESS); ValidRequests = GETPLUGINFO;	CC1
	TX70c	Processing of GETPLUGINFO fails Send(INFO,appropriate status); ValidRequests = none;	CC1

Figure 62 -- Connection client state machine: GETPLUGINFO processing

State CC70. Upon entry, the connection client generates information to be returned in the response packet (connectPktID=PLUGINFO).
Transition TX70a. A bus reset occurs. All allocated resources for handling this request are freed. The connection client's lock register is unlocked. CC_busReset() is called.
Transition TX70b. The information has been gathered. A response packet is sent. The set of valid requests is set to GETPLUGINFO.
Transition TX70c. The processing of GETPLUGINFO failed. A response packet is sent. The set of valid requests is set to none.

6.20 Consumer state machine

The consumer function is broken up into 2 state machines – one for handling large frames and one for handling small frames.

6.20.1 Large frame consumer state machine terminology

The table below explains terms used by the large frame consumer state machine.

Term	Explanation
BusReset	A 1394 bus reset occurs.
CreationEvent	A CreationEvent occurs after the plug is successfully created and the connection register is unlocked.
WriteLFPPR	Variable set True if the consumer needs to write to the connected node LargeFrameProducer register.
LFC	Consumer's private copy of LargeFrameConsumer register.
LFC'	The public plug LargeFrameConsumer register that a producer writes to.
LogErr()	Should log an error. This is implementation dependent.
Rcode	The 1394 transaction response code that should be returned if the implementation has control over the returned response code.
ReactivationEvent	A ReactivationEvent occurs after a REACT connection request is received for the plug and the connection register has been unlocked.
WriteLF_Producer-PortRegisters()	Writes to the LargeFrameProducer register and the LargeFramePageTableElement[] registers.

Table 12 – Large frame consumer state machine terminology

6.20.2 Large frame consumer state machine

Note that the logic for the conditions requires the condition for TX#a to be evaluated before the condition for TX#b, which is evaluated before the condition for TX#c, and so on.

State	Transi-tion	Condition Action	New state
CLF0:new Initialize consumer port. LFC=0; WriteLFPPR=T;	TX0a	BusReset || valid FREE request packet is received for this plug Free consumer port resources;	Exit
	TX0b	CreationEvent received from connection client state machine. 	CLF1
CLF1: Write ProducerLimit register	TX1a	BusReset RestoreState=CLF1	CLF4
	TX1b	Write of ProducerLimits register failed. LogErr();	CLF5
	TX1c	Write of ProducerLimits register completed successfully. 	CLF2

CLF2:consume
if (WriteLFPPR)
{
WriteLF_ProducerPortRegisters(LFP.sc=~LFC.sc);
WriteLFPPR=F;
}
 	
Wait for LargeFrame-Consumer register update.
	TX2a	BusReset
		RestoreState=CLF2
			CLF4

	TX2b	A valid STOP request packet is received for this plug. 	CLF5
	TX2c	WriteLF_ProducerPortRegisters() fails LogErr();	CLF5
	TX2d	LFC' updated && LFC'.sc == LFC.sc Ignore update. Rcode = resp_complete.	CLF2
	TX2e	LFC' updated && (LFC'.count >= 0) && ((LFC'.mode == LAST) || (LFC'.mode == TRUNC)) LFC=LFC'; WriteLFPPR=T; Frame received indication sent to higher layer.	CLF3
	TX2f	LFC' updated && (LFC'.count > 0) && (LFC'.mode == MORE) LFC=LFC'; WriteLFPPR=T; Frame content received Indication may be sent to higher layer.	CLF3
	TX2g	LFC' updated LogErr(); ignore update; Rcode = resp_data_err;	CLF2
	TX2h	Large frame content received. Frame content received indication may be sent to higher layer	CLF2
CLF3: Wait for a large frame segment buffer to be available	TX3a	BusReset RestoreState=CLF3	CLF4
	TX3b	A valid STOP request packet is received for this plug. 	CLF5
	TX3c	Large frame segment buffer available 	CLF2
CLF4:deactivated	TX4a	BusReset 	CLF4
	TX4b	A valid STOP request packet is received for this plug. 	CLF5
	TX4c	ReactivationEvent received from connection client state machine. 	RestoreState
	TX4d	LFC' updated || (large frame content is received) LogErr(); Ignore update; Rcode = resp_conflict_err;	CLF4
CLF5:waitFREE	TX5a	BusReset || valid FREE request packet is received for this plug Free consumer port resources	Exit

Figure 63 – Large frame consumer state machine

State CLF0. This is the initial large frame consumer state machine state when a plug is created. Variable LFC is set to 0 and variable WriteLFPPR is set true.
TransitionTX0a. A bus reset occurs or a valid FREE request packet is received for this plug. Any resources associated with the consumer port are freed. The state machine terminates.
Transition TX0b. The connection register is unlocked, allowing consumer activity to commence.

State CLF1. Upon entry, the consumer writes to the connected node ProducerLimits register.
Transition TX1a. A bus reset occurs. Variable RestoreState is set to the current state, CLF1.
Transition TX1b. The write of the ProducerLimits register fails. An error should be logged.
Transition TX1c. The write of the ProducerLimits register completes and all of the above conditions evaluate false. Variable WriteLFPPR is set true.

State CLF2. Upon entry, if WriteLFPPR is true, the consumer writes to the producer's LargeFrameProducer and, if necessary, the LargeFramePageTableElement[] registers to allow the producer to begin a large frame transfer. The consumer writes to the connected node so that LargeFrameProducer.sc-bit is set to the complement of the consumer's LargeFrameConsumer.sc bit. The consumer waits for a LargeFrameConsumer update.
Transition TX2a. A bus reset occurs. Variable RestoreState is set to the current state, CLF2.
Transition TX2b. A valid STOP request is received for this plug.
Transition TX2c. WriteLF_ProducerPortRegisters fails. An error should be logged.
Transition TX2d. A LargeFrameConsumer update occurs, but the sc-value is incorrect. This update is ignored. A response code resp_complete should be returned.
Transition TX2e. A LargeFrameConsumer update occurs and the producer has completed a large frame transfer. The consumer copies the new LargeFrameConsumer value. A frame-received indication shall be sent to a higher layer.
Transition TX2f. A LargeFrameConsumer update occurs and the producer has transferred part of a frame, but not the end of a frame. The consumer copies the new LargeFrameConsumer value. A consumer may send an indication to a higher layer so the part of the frame that has been transferred may be processed.
Transition TX2g. A LargeFrameConsumer update occurs, but all of the above conditions have tested false. An error should be logged. The update is ignored. A response code = resp_data_err should be returned.
Transition TX2h. Frame content is received. Some implementations may receive an interrupt on each transfer of large frame content. A consumer may send an indication to a higher layer so the part of the frame that has been transferred may be processed.

State CLF3. The consumer waits for an available consumer segment buffer.
Transition TX3a. A bus reset occurs.
Transition TX3b. A valid STOP request packet is received.
Transition TX3c. A large frame consumer segment buffer becomes available.

State CLF4. This is the deactivated state.
Transition TX4a. A bus reset occurs.
Transition TX4b. A valid STOP request packet is received.
Transition TX4c. A ReactivationEvent for this plug is received from the connection client state machine.
Transition TX4d. A LargeFrameConsumer update occurs or large frame content is received. An error should be logged. The plug activity should be ignored. A response code = resp_conflict_err should be returned.

State CLF5. The consumer state machine waits for a FREE packet to be received.
Transition TX5a. A bus reset occurs or a FREE packet is received.

6.20.3 Small frame consumer state machine terminology

The table below explains terms used by the small frame consumer state machine.

Term	Explanation
BusReset	A 1394 bus reset occurs.
CreationEvent	A CreationEvent occurs after the plug is successfully created and the connection register is unlocked.
WriteSFPPR	Variable set True if the consumer needs to update the connected node SmallFrameProducer register.
LogErr()	Should log an error. This is implementation dependent.
Rcode	The 1394 transaction response code that should be returned if the implementation has control over the returned response code.
ReactivationEvent	A ReactivationEvent occurs when a REACT connect request packet has been received for this plug and the connection register has been unlocked.
SFC	Consumer's private copy of the SmallFrameConsumer register.
SFC'	The public plug SmallFrameConsumer register that a producer writes to.
WriteSF_Producer-PortRegisters()	Writes to the SmallFramePageTableElement register and the SmallFrameProducer register.

Table 13 -- Small frame consumer state machine terminology

6.20.4 Small frame consumer state machine

Stfate	Transi-tion	Condition Action	New state
CSF0: new Initialize consumer port. SFC = 0; WriteSFPPR = T;	TX0a	BusReset || valid FREE request packet is received for this plug Free consumer port resources;	Exit
	TX0b	CreationEvent received from connection client state machine. 	CSF1

CSF1:consume
If (WriteSFPPR)
{
WriteSF_ProducerPortRegisters(SFP.sc=~SFC.sc);
WriteSFPPR=F;
}
 	
Wait for SmallFrame-Consumer register update.
	TX1a	BusReset
		RestoreState=CSF1
			CSF3

	TX1b	A valid STOP request packet is received for this plug. 	CSF4
	TX1c	WriteSF_ProducerPortRegisters() fails LogErr();	CSF4
	TX1d	SFC' updated && SFC'.sc == SFC.sc Ignore update; Rcode = resp_complete	CSF1
	TX1e	SFC' updated && SFC'.mode == 1 (SFB_FULL) SFC=SFC'; WriteSFPPR = T;	CSF2
	TX1f	SFC' updated LogErr(); ignore update; Rcode = resp_data_err	CSF1
	TX1g	Small frame content received. Frame received indication sent to higher layer.	CSF1
CSF2: Wait for a small frame segment buffer to be available	TX2a	BusReset RestoreState=CSF2	CSF3
	TX2b	A valid STOP request packet is received for this plug. 	CSF4
	TX2c	Small frame segment buffer available 	CSF1
CSF3:deactivated	TX3a	BusReset 	CSF3
	TX3b	A valid STOP request packet is received for this plug. 	CSF4
	TX3c	ReactivationEvent received from connection client state machine 	RestoreState
	TX3d	SFC' updated || (small frame received) LogErr(); Ignore update; Rcode = resp_conflict_err;	CSF3
CSF4:waitFREE	TX4a	A FREE request packet is received || BusReset Free consumer port resources	Exit

Figure 64 - Small frame consumer state machine

State CSF0. This is the initial small frame consumer state machine state when a plug is created. Variable SFC is set to 0 and variable WriteSFPPR is set true.
TransitionTX0a. A FREE request packet is received for this plug, or a bus reset occurs. Any resources associated with the consumer port are freed. The consumer state machine terminates.
Transition TX0b. The connection register is unlocked, allowing consumer activity to commence. A consumer may wait for a read request from a higher layer before transitioning to the next state.

State CSF1. Upon entry, if WriteSFPPR is true, the consumer writes to the producer's SmallFrameProducer and, if necessary, the SmallFramePageTableElement registers at this time to allow the producer to begin a small frame transfer. The consumer writes to the connected node so that SmallFrameProducer.sc-bit is set to the complement of the consumer's SmallFrameConsumer.sc bit The consumer waits for small frames and/or a SmallFrameConsumer register update.
Transition TX1a. A bus reset occurs.
Transition TX1b. A valid STOP request is received for this plug.

Transition TX1c. WriteSF_ProducerPortRegisters fails. An error should be logged.
Transition TX1d. A SmallFrameConsumer update occurs, but the sc-value is incorrect. This update is ignored. A response code = resp_complete should be returned.

Transition TX1e. A SmallFrameConsumer update occurs and the new value of the SmallFrameConsumer.mode-bit is set to 1 indicate the small frame consumer segment buffer is exhausted. The consumer copies the new SmallFrameConsumer value. Variable WriteSFPPR is set true.
Transition TX1f. A SmallFrameConsumer update occurs and all of the above conditions tested false. An error should be logged. The update is ignored. A response code = resp_data_err should be returned.
Transition TX1g. A small frame is received. A frame-received indication shall be sent to a higher layer.

State CSF2. The consumer waits for an available small frame segment buffer.
Transition TX2a. A bus reset occurs.
Transition TX2b. A valid STOP request packet is received.
Transition TX2c. A small frame segment buffer becomes available.

State CSF3. This is the deactivated state. The consumer state machine waits for a REACT packet to be sent.
Transition TX3a. A bus reset occurs.
Transition TX3b. A valid STOP request packet is received.
Transition TX3c. A ReactivationEvent for this plug is received from the connection client state machine.
Transition TX3d. A SmallFrameConsumer update occurs or a small frame is received. An error should be logged. The plug activity should be ignored. A response code = resp_conflict_err should be returned.

State CSF4. The consumer state machine waits for a FREE packet to be received.
Transition TX4a. A bus reset occurs or a FREE packet is received.

6.21 Producer state machines

The producer function is broken up into 2 state machines – one for handling large frames and one for handling small frames.

6.21.1 Large frame producer state machine terminology

Term	Explanation
BusReset	A 1394 bus reset occurs.
fp	Pointer to application data structure returned from WaitForLargeFrameContent(), WaitForSmallFrameContent(). fp->data = pointer to data buffer fp->SF = TRUE if fp->data represents all of a frame and it will fit and be transmitted in a single 1394 write request. fp->MODE = LAST if fp->data represents last part of a complete frame fp->MODE = MORE if fp->data does not represent the last part of a frame. fp->MODE = TRUNC if fp->data represents the end of a truncated frame. fp->size = size (in bytes) for fp->data buffer to be sent. fp->residue = remaining number of bytes to send
FutureLFC	A producer variable holding a LargeFrameConsumer value to be written later to the LargeFrameConsumer register on the connected node.
LF_offset	Large frame buffer offset. The producer uses this to keep track of the total number of bytes sent to the consumer's large frame buffer space specified in the PageTableElement[] registers.
LF_PTE	Current LargeFramePageTableElement[] register in use.
LF_PTE_offset	Number of bytes sent to the space in the current LargeFramePageTableElement.
LFP	Producer's private copy of LargeFrameProducer register.
LFP'	The public LargeFrameProducer register that a consumer writes to.
LogErr()	Should log an error. This is implementation dependent.
Rcode	The 1394 transaction response code that should be returned if the implementation has control over the returned response code.
ReactivationEvent	A ReactivationEvent occurs when a REACT connection request is received for the plug and the connection register has been unlocked.
RestoreState	Variable holding the producer state machine state prior to bus reset.
WaitForLargeFrame-Content()	WaitForLargeFrameContent() waits for new large frame content to write. The amount of frame content buffered is implementation dependent. If WaitForLargeFrameContent() is called and there is still some residual frame content to send (fp->residue != 0) WaitForLargeFrameContent() returns immediately so the residue is sent. WaitForLargeFrameContent() returns if a fp->MODE == LAST or fp->MODE == TRUNC indication is received, even if there is no frame content to be sent.
WriteLargeFrame(fp, PTE, offset, nbytes);	Writes large frame content to the space described in the LargeFramePageTableElement[] register array. The fp variable points to the frame content. Variable PTE is the PageTableElement register. Variable offset is the offset from the current LargeFramePageTableElement destination_offset where writes are to begin. Variable nbytes specifies the number of bytes to write.
WriteLFC()	Write to the connected node LargeFrameConsumer register.

Table 14 -- Large frame producer state machine terminology

6.21.2 Large frame producer state machine, states LFP0 – LFP4

State	Tran-sition	Condition Action	New state
LFP0: new	TX0a	 LFP = 0	LFP1
LFP1: Wait for LargeFrame-Producer update	TX1a	BusReset RestoreState=LFP1;	LFP5
	TX1b	A valid STOP request packet is received for this plug. 	LFP6
	TX1c	(LFP' updated && LFP'.sc == LFP.sc) Ignore update; Rcode = resp_complete	LFP1
	TX1d	LFP' updated && (LFP'.run == 1) && (LFP'count > 0) LFP = LFP'; LF_offset = 0; LF_PTE = LargeFramePageTableElement[0]; LF_PTE_offset = 0; Rcode = resp_complete	LFP2
	TX1e	LFP' updated and all above conditions evaluated false LogErr(); ignore update; Rcode = resp_type_error	LFP1
LFP2: WaitForLarge-FrameContent()	TX2a	BusReset RestoreState=LFP2;	LFP5
	TX2b	A valid STOP request packet is received for this plug. 	LFP6
	TX2c	Large frame write request received && LF_PTE_offset + fp->residue > LF_PTE.length PTE = LF_PTE, offset = LF_PTE_offset; nbytes = LF_PTE.length – LF_PTE_offset	LFP3
	TX2d	Large frame write request received PTE=LF_PTE; offset=LF_PTE_offset; nbytes=fp->residue	LFP3
	TX2e	LFP' updated LogErr(); Ignore update; Rcode = resp_type_error;	LFP2
LFP3: WriteLarge-Frame(fp, PTE, offset, nbytes); LF_offset += nbytes; LF_PTE_offset += nbytes; fp->residue -= nbytes;	TX3a	BusReset RestoreState=LFP3;	LFP5
	TX3b	A valid STOP request packet is received for this plug. 	LFP6
	TX3c	WriteLargeFrame() fails 	LFP6
	TX3d	fp->END_OF_FRAME && fp->residue == 0 FutureLFC.sc=LFP.sc; FutureLFC.mode=LAST; FutureLFC.count=LF_offset; Indication(frame sent);	LFP4
	TX3e	LF_offset == LFP.count && fp->residue FutureLFC.sc=LFP.sc; FutureLFC.mode=MORE; FutureLFC.count=LF_offset;	LFP4
	TX3f	fp->residue LF_PTE = next LargeFramePageTableElement; LF_PTE_offset = 0;	LFP2
	TX3g	(LF_offset != LFP.count) && (LF_PTE_offset == LF_PTE.length) LF_PTE = next LargeFramePageTableElement; LF_PTE_offset = 0; Indication(frame content sent);	LFP2
	TX3h	LF_offset != LFP.count Indication(frame content sent);	LFP2
	TX3k	WriteLargeFrame() finished and all above conditions evaluated to false FutureLFC.sc=LFP.sc; FutureLFC.mode=MORE; FutureLFC.count=LF_offset; Indication(frame content sent);	LFP2
	TX3m	LFP' updated LogErr(); Rcode = resp_type_error	LFP3

Figure 65 -- Large frame producer state machine, states LFP0-LFP3

6.21.3 Large frame producer state machine, states LFP4 – LFP7

State	Tran-sition	Condition Action	New state
LFP4: WriteLFC(FutureLFC);	TX4a	BusReset RestoreState=LFP4;	LFP5
	TX4b	A valid STOP request received 	LFP6
	TX4c	WriteLFC succeeds 	LFP1
	TX4d	WriteLFC fails LogErr();	LFP6
	TX4e	LFP' updated (before WriteLFC completes) LogErr(); Rcode = resp_type_error;	LFP4
LFP5:deactivtd LFP.run = 0;	TX5a	BusReset 	LFP5
	TX5b	A valid STOP request received 	LFP6
	TX5c	LFP' updated Ignore update; Rcode = resp_complete	LFP5
	TX5d	ReactivationEvent received LFP.run = 1	RestoreState
LFP6: Stop	TX6a	BusReset 	LFP7
	TX6b	LFP' updated Ignore update; rcode = resp_complete	LFP6
	TX6c	All large frame transfers have stopped 	LFP7
LFP7:waitFREE	TX7a	BusReset || Valid FREE packet received 	Exit
	TX7b	LFP' updated Ignore update; rcode = resp_complete	LFP7

Figure 66 -- Large frame producer state machine, states LFP4-LFP7

State LFP0. The initial large frame producer state machine state.
Transition TX0a. The producer port is initialized. The LargeFrameProducer register is set to zero.

State LFP1. Producer state machine waits for a LargeFrameProducer register update.
Transition TX1a. A bus reset occurs. Variable RestoreState is set to the current state, LFP1.
Transition TX1b. A valid STOP request is received.
Transition TX1c. An update of the LargeFrameProducer register occurs but the sc-bit value is the same as the current sc-bit value. A response code = resp_complete should be returned.
Transition TX1d. An update of the LargeFrameProducer register occurs and the run-bit is set and the count is > 0. The producer's private copy of the LargeFrameProducer register is set to the updated value. Variable LF_PTE is set to LargeFramePageTableElement[0] and variables LF_offset and LF_PTE_offset are set to 0. A response code = resp_complete should be returned.
Transition TX1e. The LargeFrameProducer register has been updated and all above conditions have tested false. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State LFP2. State machine waits for large frame content to transfer.
Transition TX2a. A bus reset occurs. Variable RestoreState is set to the current state, LFP2.
Transition TX2b. A valid STOP request is received.

Transition TX2c. A request from a higher layer to transfer large frame content is received and the frame content will not fit into the current LargeFramePageTableElement[] space. Variable PTE is set to LF_PTE, PTE_offset is set to LF_PTE_offset, and nbytes is set to exactly fill up the current LargeFramePageTableElement[] register.
Transition TX2d. A request from a higher layer to transfer large frame content is received and the frame content will fit into the current LargeFramePageTableElement[] space. Variable PTE is set to LF_PTE, PTE_offset is set to LF_PTE_offset, and nbytes is set so that all of the frame content is transferred.
Transition TX2e. An unexpected LargeFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State LFP3. Upon entry, the state machine writes the frame content. When done, variables LF_offset and LF_PTE_offset are incremented by the number of bytes sent. Variable fp->residue is reduced by the number of bytes sent.
Transition TX3a. A bus reset occurs. Variable RestoreState is set to the current state, LFP3.
Transition TX3b. A valid STOP request is received.
Transition TX3c. WriteLargeFrame() fails for some reason. An error should be logged.
Transition TX3d. WriteLargeFrame() completes and the large frame content sent represents the last of the frame and there is no more frame content to be sent. Variable FutureLFC.sc is set to LargeFrameProducer.sc. Variable FutureLFC.mode is set to LAST. Variable FutureLFC.count is set to LF_offset. An indication to the higher layer that the frame has been sent.
Transition TX3e. WriteLargeFrame() completes, all above conditions evaluated false, and there is more large frame content to be sent, but the space described by the LargeFramePageTableElement[] registers is exhausted. Variable FutureLFC.sc is set to LargeFrameProducer.sc. Variable FutureLFC.mode is set to MORE. Variable FutureLFC.count is set to LF_offset.
Transition TX3f. WriteLargeFrame() completes, all above conditions evaluated false, and there is more large frame content to be sent. Variable LF_PTE is set to the next adjacent LargeFramePageTableElement[] register and LF_PTE_offset is set to 0.
Transition TX3g. WriteLargeFrame() completes, all above conditions evaluated false, and there is no space remaining in the space described by the current LargeFramePageTableElement[]. Variable LF_PTE is set to the next adjacent LargeFramePageTableElement[] and variable LF_PTE_offset is reset to 0. An indication is returned that the frame content was sent.
Transition TX3h. WriteLargeFrame() completes, all above conditions evaluated false, and variable LF_offset != LF_LFP.count.
Transition TX3k. WriteLargeFrame() completes and all above conditions evaluated false. An indication is returned that the frame content was sent. Variable FutureLFC.sc is set to LargeFrameProducer.sc. Variable FutureLFC.mode is set to MORE. Variable FutureLFC.count is set to LF_offset.
Transition TX3m. An unexpected LargeFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State LFP4. Upon entry, the state machine writes to the connected node LargeFrameConsumer register.
Transition TX4a. A bus reset occurs. Variable RestoreState is set to the current state, LFP4.
Transition TX4b. A valid STOP request is received.
Transition TX4c. WriteLFC() succeeds.
Transition TX4d. WriteLFC() fails. An error should be logged.
Transition TX4e. An unexpected LargeFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State LFP5. This is the deactivated state. Upon entry, the LFP.sc and LFP.run bits are cleared.
Transition TX5a. A bus reset occurs.
Transition TX5b. A valid STOP request is received.
Transition TX5c. An unexpected LargeFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

Transition TX5d. A ReactivationEvent for this plug is received from the connection client state machine.

State LFP6. The producer shall not initiate the sending of any more frame content. The producer should abort any frame content transfers that are queued.
Transition TX6a. A bus reset occurs.
Transition TX6b. A LargeFrameProducer register update occurs.
Transition TX6c. All frame transfers have stopped.

State P7. The producer state machine waits for a FREE packet to be received.
Transition TX7a. A bus reset occurs or a FREE packet is received. The producer frees producer port resources. The state machine terminates.
Transition TX7b. An update of the LargeFrameProducer register occurs. The update is ignored. A response code = resp_complete should be returned.

6.21.4 Small frame producer state machine terminology

The table below explains terms used by the small frame producer state machine.

Term	Explanation
BusReset	A 1394 bus reset occurs.
Fp	Pointer to application data structure returned from WaitForLargeFrameContent(), WaitForSmallFrameContent(). fp->data = pointer to data buffer fp->SF = TRUE if fp->data represents all of a frame and it will fit and be transmitted in a single 1394 write request. fp->MODE = LAST if fp->data represents last part of a complete frame fp->MODE = MORE if fp->data does not represent the last part of a frame. fp->MODE = TRUNC if fp->data represents the end of a truncated frame. fp->size = size (in bytes) for fp->data buffer to be sent. fp->residue = remaining number of bytes to send
LogErr()	Should log an error. This is implementation dependent.
NF	Number of small frames sent since last SmallFrameProducer register update
Rcode	The response code that should be returned if the implementation has control over the returned response code.
ReactivationEvent	A ReactivationEvent occurs when a REACT connection request is received for the plug and the connection register has been unlocked.
RestoreState	Variable holding the producer state machine state prior to bus reset.
SF_offset	Small frame space offset. The producer uses this to keep track of how many bytes have been sent to the consumer's small frame space.
SF_PTE	SmallFramePageTableElement register
SFP	Producer's private copy of the SmallFrameProducer register.
SFP'	The public SmallFrameProducer register that a consumer writes to.
WaitForSmallFrame-Content()	WaitForSmallFrameContent() waits, if necessary, for new small frame content to write. If a bus reset occurs after WaitForSmallFrameContent() returned with a small frame to send but before the small frame was sent, WaitForSmallFrameContent() returns immediately so the small frame is sent.
WriteSFC()	Write to the connected node SmallFrameConsumer register.
WriteSmallFrame(fp, offset, nbytes)	Writes a small frame to the space described by the SmallFramePageTableElement. The fp variable points to the frame content. Variable PTE is the PageTableElement register. Variable offset is the offset from the PageTableElement destination_offset where writes are to begin. Variable nbytes specifies the number of bytes to write.

Table 15 -- Small frame producer state machine terminology

6.21.5 Small frame producer state machine, states SFP0 – SFP4

State	Tran-sition	Condition Action	New state
SFP0: new	TX0a	 SFP = 0	SFP1
SFP1: Wait for SmallFrame-Producer update	TX1a	BusReset RestoreState=SFP1;	SFP5
	TX1b	A valid STOP request packet is received for this plug. 	SFP6
	TX1c	(SFP' updated && SFP'.sc == SFP.sc) Ignore update; Rcode = resp_complete	SFP1
	TX1d	SFP' updated && (SFP'.run == 1) && (SFP'count > 0) SFP = SFP'; SF_offset = 0; SF_PTE = SmallFramePageTableElement; NF = 0; Rcode = resp_complete	SFP2
	TX1e	SFP' updated and all above conditions evaluated false LogErr(); ignore update; Rcode = resp_type_error	SFP1
SFP2: WaitForSmall-FrameContent()	TX2a	BusReset RestoreState=SFP2;	SFP5
	TX2b	A valid STOP request packet is received for this plug. 	SFP6
	TX2c	Small frame write request received && SF_offset + fp->residue > SF_PTE..length 	SFP4
	TX2d	Small frame write request received PTE=SF_PTE; offset=SF_offset; nbytes=fp->residue	SFP3
	TX2e	SFP' updated LogErr(); Ignore update; Rcode = resp_type_error;	SFP2
SFP3: WriteSmall-Frame(fp, PTE, offset, nbytes); SF_offset += nbytes; fp->residue -= nbytes; NF++;	TX3a	BusReset RestoreState=SFP3;	SFP5
	TX3b	A valid STOP request packet is received for this plug. 	SFP6
	TX3c	WriteSmallFrame() fails 	SFP6
	TX3d	WriteSmallFrame() completes && (SF_offset == SF_PTE.length) || (NF == SFP.maxSmallFrameCount) Indication(small frame sent);	SFP4
	TX3e	WriteSmallFrame() completes 	SFP2
	TX3f	SFP' updated LogErr(); Rcode = resp_type_error	SFP3

Figure 67 -- Small frame producer state machine, states SFP0-SFP3

6.21.6 Small frame producer state machine, states SFP4 – SFP7

State	Tran-sition	Condition Action	New state
SFP4: WriteSFC(SFC.sc=SFP.sc, SFC.mode=1);	TX4a	BusReset RestoreState=SFP4;	SFP5
	TX4b	A valid STOP request received 	SFP6
	TX4c	WriteSFC succeeds 	SFP1
	TX4d	WriteSFC fails LogErr();	SFP6
	TX4e	SFP' updated (before WriteSFC completes) LogErr(); Rcode = resp_type_error;	SFP4
SFP5:deactivtd SFP.run = 0;	TX5a	BusReset 	SFP5
	TX5b	A valid STOP request received 	SFP6
	TX5c	SFP' updated Ignore update; Rcode = resp_complete	SFP5
	TX5d	ReactivationEvent received SFP.run = 1;	RestoreState
SFP6: Stop	TX6a	BusReset 	SFP7
	TX6b	SFP' updated Ignore update; rcode = resp_complete	SFP6
	TX6c	All Small frame transfers have stopped 	SFP7
SFP7:waitFREE	TX7a	BusReset || valid FREE packet received 	Exit
	TX7b	SFP' updated Ignore update; rcode = resp_complete	SFP7

Figure 68 -- Small frame producer state machine, states SFP4-SFP7

State SFP0. The initial Small frame producer state machine state.
Transition TX0a. The producer port is initialized. The SmallFrameProducer register is set to zero.

State SFP1. Producer state machine waits for a SmallFrameProducer register update.
Transition TX1a. A bus reset occurs. Variable RestoreState is set to the current state, SFP1.
Transition TX1b. A valid STOP request is received.
Transition TX1c. An update of the SmallFrameProducer register occurs but the sc-bit value is the same as the current sc-bit value. A response code = resp_complete should be returned.
Transition TX1d. An update of the SmallFrameProducer register occurs and the run-bit is set and the count is > 0. The producer's private copy of the SmallFrameProducer register is set to the updated value. Variable SF_PTE is set to SmallFramePageTableElement[0] and variables SF_offset and SF_PTE_offset are set to 0. A response code = resp_complete should be returned.
Transition TX1e. The SmallFrameProducer register has been updated and all above conditions have tested false. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State SFP2. State machine waits for Small frame content to transfer.
Transition TX2a. A bus reset occurs. Variable RestoreState is set to the current state, SFP2.

Transition TX2b. A valid STOP request is received.
Transition TX2c. A request from a higher layer to transfer a small frame is received and the frame content will not fit into the space described by the SmallFramePageTableElement. None of the small frame content is transferred.
Transition TX2d. A request from a higher layer to transfer a small frame is received and the frame content will fit into the space described by the SmallFramePageTableElement. Variable PTE is set to SF_PTE, PTE_offset is set to SF_offset, and nbytes is set so that all of the frame content is transferred.
Transition TX2e. An unexpected SmallFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State SFP3. Upon entry, the state machine writes the frame content. When done, variables SF_offset and SF_PTE_offset are incremented by the number of bytes sent. Variable fp->residue is reduced by the number of bytes sent.
Transition TX3a. A bus reset occurs. Variable RestoreState is set to the current state, SFP3.
Transition TX3b. A valid STOP request is received.
Transition TX3c. WriteSmallFrame() fails for some reason. An error should be logged.
Transition TX3d. WriteSmallFrame() completes and the small frame space is exhausted or the maximum number of small frames has been sent. An indication to the higher layer that the frame has been sent.
Transition TX3e. WriteSmallFrame() completes and all above conditions evaluated false.
Transition TX3f. An unexpected SmallFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State SFP4. Upon entry, the state machine writes to the connected node SmallFrameConsumer register.
Transition TX4a. A bus reset occurs. Variable RestoreState is set to the current state, SFP4.
Transition TX4b. A valid STOP request is received.
Transition TX4c. WriteSFC() succeeds.
Transition TX4d. WriteSFC() fails. An error should be logged.
Transition TX4e. An unexpected SmallFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.

State SFP5. This is the deactivated state. Upon entry, the SFP.sc and SFP.run bits are cleared.
Transition TX5a. A bus reset occurs.
Transition TX5b. A valid STOP request is received.
Transition TX5c. An unexpected SmallFrameProducer register update occurs. An error should be logged. The update shall be ignored. A response code = resp_type_error should be returned.
Transition TX5d. A ReactivationEvent for this plug is received from the connection client state machine.

State SFP6. The producer shall not initiate the sending of any more small frames. The producer should abort any frame transfers that are queued.
Transition TX6a. A bus reset occurs.
Transition TX6b. A SmallFrameProducer register update occurs.
Transition TX6c. All frame transfers have stopped.

State P7. The producer state machine waits for a FREE packet to be received.
Transition TX7a. A bus reset occurs or a FREE packet is received. The producer frees producer port resources. The state machine terminates.
Transition TX7b. An update of the SmallFrameProducer register occurs. The update is ignored. A response code = resp_complete should be returned.

7. IICP services (informative)

IICP layer services are provided at the interface between the IICP layer and higher layers.

Row	Service	Layer communicated with	Purpose of service
1	IICP control request	From higher layer	This service performs one or more of the following: • Initialize the IICP layer. • Configure the IICP layer.
2	IICP open request	From higher layer	Causes a connection sequence to be issued.
3	IICP close request	From higher layer	Causes a connection to be torn down and freed.
4	IICP write data frame request	From higher layer	Causes a data frame to be sent.
5	IICP write control frame request	From higher layer	Causes a control frame to be sent.
6	IICP read data frame request	From higher layer	Causes an update of the data port ProducerMode register.
7	IICP CREQ1 indication	To higher layer	Indicate the reception of a CREQ1 packet from another node..
8	IICP connection established indication	To higher layer	Indicate the completion of a connection sequence initiated by another node.
9	IICP stop indication	To higher layer	Indicate the reception of a STOP packet
10	IICP error indication	To higher layer	Indicate an IICP layer error or lower level protocol error.
11	IICP control frame received indication	To higher layer	Indicate reception and convey contents of control frame to higher layer.

7.1 IICP control request

The higher layer uses this service to perform one or more of the following:
1. Initialize the IICP layer.
2. Configure the IICP layer.

Ever power-on sequence should result in control requests to initialize the layer and to configure the IICP layer.

7.1.1 Initialization of IICP layer

This operation results in the initialization of the connection manager state machine (if the implementation can perform as a connection manager) and the connection client state machine. This operation would also initialize the connection lock register and map the 512-byte connection register to 1394 space.

7.1.2 Configure the IICP layer

This operation would communicate any tunable parameters to the IICP layer. Examples of tunable parameters are:
1. Maximum and minimum sizes for buffers.
2. Maximum sizes for scatter/gather page tables.

This would also allow registration of callback functions for certain events that a higher layer would be interested in knowing about. Suggested callback functions an implementation may provide are:
1. IICP_CREQ1_callback (…)
2. IICP_connectionEstablished_callback (…)
3. IICP_STOP_callback (…)
4. IICP_err_callback (…)

7.1.2.1 IICP_CREQ1_callback (…)

Notifies higher layer of a new connection. The following parameters from the CREQ1 packet are communicated to the higher layer.

1. CommandSet
2. ConnectionParameters

The higher layer may return the following information to the IICP layer:

1. Acceptance or rejection of the connection request.
2. Whether the connected node is required to write sequentially.
3. The size of data frames that the higher layer may produce.
4. The size of control frames that the higher layer may produce.

After receiving the information from the higher layer, the IICP layer formulates the connection response (CRESP1).

If the higher layer has not registered an IICP_CREQ1_callback(…), the IICP layer should proceed with default parameters.

If the higher layer has registered an IICP_CREQ1_callback(…) but only returns partial information, the IICP layer should proceed with default parameters as appropriate.

7.1.2.2 IICP_connectionEstablished_callback (…)

Notifies higher layer of the completion of a connection sequence initiated by another node. This service communicates to a higher layer that it is now permissible to perform reads and writes through this new connection. The following parameters from the CREQ2 packet is communicated to the higher layer.

1. The data frame size from the connected node.
2. The control frame size from the connected node.
3. Whether the connected node requires sequential writes.
4. A handle to the newly created connection.

7.1.2.3 IICP_STOP_callback (…)

Notifies higher layer of the cessation of plug activity.

7.1.2.4 IICP_err_callback (…)

Notifies higher layer of an IICP error or lower protocol error. A higher layer may log the error to a file, to an I/O port for printing, or to a display. An example of the use of IICP_err_callback() is if a plug fails reactivation after a bus reset.

7.2 IICP open request

This service, available on nodes that have connection manager capability, issues a connection sequence and establishes a plug connection between two IICP devices. The higher layer provides the following information so the IICP layer can perform the request:

1. command_set_spec_id, command_set, and command_set_details.
2. connectionParameters.
3. Some form of node identification for the node to be connected.
4. The data frame size.
5. The control frame size.
6. Whether sequential writes are required.

When the connection sequence has completed, the following information is provided to the higher layer:

1. Success or failure of establishing the connection.

2. Whether sequential writes are required when writing to the connected node.
3. The size of data frames from the connected node.
4. The size of control frames from the connected node.

7.3 IICP close request

This service, available on nodes that have connection manager capability, issues a connection sequence to stop a plug.

7.4 IICP write data frame request

This service results in a data frame or a part of a data frame being sent to the connected node. Example, informative, pseudo-code is shown below.

```
IICP_write(…, bool END_OF_FRAME, …)
{
   bool sent = false;
   bool isSmallFrame = true;

   if ( request size > 512 bytes
     || request does not represent complete frame ) {
      isSmallFrame = false;
   }
   if ( small frame producer state == SFP2
         && isSmallFrame
         && END_OF_FRAME ) {
      sent = SmallFrameWriteRequest(…); // might not be sent if
                                        // not enough space
   }
   if ( !sent ) { // send as large frame
      block until large frame producer state == LFP2;
      LargeFrameWriteRequest(…);
   }
}
```

7.5 IICP write control frame request

This service results in a control frame or part of a control frame being sent to the connected node. This service may provide a mechanism for higher layer protocols (for example, IICP488) to send asynchronous stream trigger packets.

7.6 IICP read data frame request

This service results in the programming of the data port ProducerMode register on the connected node. The connected node may then begin sending data frame(s).

The IICP layer blocks until the read request is satisfied. The criteria for returning from a read request are implementation and higher level protocol dependent.

7.7 IICP CREQ1 indication

If a callback has been registered (see the IICP control service), the callback function is called with implementation-dependent parameters.

7.8 IICP connection established indication

If a callback has been registered (see the IICP control service), the callback function is called with implementation-dependent parameters.

7.9 IICP stop indication

If a callback has been registered (see the IICP control service), the callback function is called with implementation-dependent parameters.

7.10 IICP err indication

If a callback has been registered (see the IICP control service), the callback function is called with implementation-dependent parameters.

7.11 IICP control frame received indication

This indication occurs when an IICP control frame is received. This indication may be a callback function that has been set up by the higher layer. A callback may have been set up in either of the following ways:

1) Via a parameter in the IICP open request.
2) Via the IICP connection established indication callback parameters.

8. Error recovery

This section further enumerates methods and practices to ensure error free IICP communications.

8.1 Application-level retries

The resources associated with plug-visible addresses are designed to be idempotent, in that the effects of single and duplicated writes are the same. For this reason, transaction faults observed by the requester (but not necessarily observed the responder) can be safely retried by the plug-level application.

1394 transaction-level errors expected to be retried include the following:

- Response timeout. The response does not return within the SPLIT_TIMEOUT specified timeout.
- Response errors. The response returns with either of the following response code (rcode) values:
  1) resp_data_error. This can be indicative of a transient transmission error.
  2) resp_conflict_error. This can be indicative of a transient bridge congestion condition.

  The same expectations apply to unified transactions completed with ack_data_error or ack_conflict_error indications (rather than resp_data_error or resp_conflict_error).

If another transaction-level response is returned, or if a reasonable number of retries fail, an error should be logged. The relevant state machine shows the behavior to follow when an error occurs.

8.2 1394 bus resets

1394 bus resets will occur on busses when devices are powered on and off or when 1394 cabling is changed. After a bus reset, a node shall deactivate all IICP connections. A reactivation request from the node that instantiated the connection is required to reactivate the connection. See section 6.13 above.

8.2.1 Bus reset while connection registers are locked

Any plug and/or plug resources in the process of being created may be freed. After a bus reset, the connection manager is required to send all connection requests over again.

8.2.2 Bus reset during updates of plug fields

A bus reset may occur while a node is updating the public memory plug fields of another node. Any bus reset that occurs prior to a final ack complete shall result in the update being re-sent.

8.2.3 Bus reset while transferring data

A 1394 bus reset can occur while transferring data. When this occurs, an IICP plug shall go to the deactivated state until a reactivation occurs. Following a reactivation (of both nodes involved in the connection), any data that was being sent that was interrupted shall be resent.

To further clarify, after a reactivation of both sides of a connection, a producer shall retransmit all outstanding write requests that were in the process of being written to a consumer. A transaction shall be considered outstanding if a write response has not been received and an ack-complete successfully sent from the producer to the consumer.

Consumers shall be tolerant of receiving duplicate write packets.

8.2.4 Duplicate writes

If a consumer receives a write with a destination offset identical to a previous write that was successfully processed, the consumer shall accept the packet and either discard the redundant write or process the write.