IVI-6.1: IVI High-Speed LAN Instrument Protocol (HiSLIP)

Apr 23, 2020
Revision 2.0

© Copyright 2020 IVI Foundation.
All Rights Reserved.

Important Information

The IVI-6.1: High-Speed LAN Instrument Protocol Specification is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at www.ivifoundation.org.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through the web site at www.ivifoundation.org.

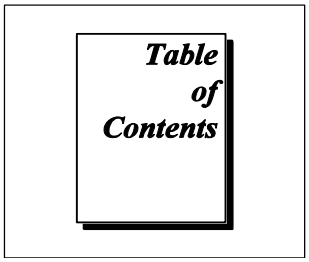
Warranty

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

Trademarks

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

No investigation has been made of common-law trademark rights in any work.



IVI HiSLIP Revision History

This section is an overview of the revision history of the IVI HiSLIP specification.

Table 1. IVI HiSLIP Class Specification Revisions

Status	Action
Revision 1.0	First version of specification.
Revision 1.1	Changes: - Incremented spec revision to 1.1 - Clarified that the protocol can be run at port other than IANA assigned port (4880) - Clarified that the sub-address is limited by VISA, but not the HiSLIP protocol - Added an observation regarding the fact that since an *AsyncUnlockResponse* is sent in response to both a lock and an unlock, a late response after a client IO timeout needs to be handled carefully, and that the timing of the unlock response is up to the server (see section 4.5.1)
Revision 2.0	Changes: - Incremented spec revision to 2.0 - Incremented protocol version to 2.0 - Added support of capabilities - Added capability Secure Connection - Editorial

1 Overview of the IVI HiSLIP Specification

HiSLIP (High Speed LAN Instrument Protocol) is a protocol for TCP-based instrument control that provides the capabilities of conventional test and measurement protocols with minimal impact to performance. The HiSLIP protocol includes:

- Device clear
- Instrument status reporting with message available calculation per IEEE Std 488.2
- Instrument remote/local status control
- Instrument locking
- Service Request from the instrument to the client
- End message
- *Message exchange protocol interrupted* error detection
- Encrypted connections
- Client and server authentication

1.1 IVI HiSLIP Overview

HiSLIP creates two TCP connections to the same server port[1] referred to as the synchronous channel and asynchronous channel. HiSLIP sends packetized messages between the client and server on both channels.

The synchronous channel carries normal bi-directional ASCII command traffic (such as SCPI) and synchronous GPIB-like meta-messages (such as END and trigger). Generally, both the client and server can leave messages in the synchronous buffers and execute them in a synchronous fashion.

The asynchronous channel carries GPIB-like meta-messages that need to be handled independently of the data path (such as device clear and service request). Generally, both the client and server need to treat asynchronous messages as higher priority and act on them before messages from the synchronous channel.

The HiSLIP protocol permits multiple virtual instruments at a single port at a given IP address. When the connection is initialized, the client specifies a sub-address that designates the specific virtual instrument to be associated with this connection. The protocol does not associate any aspects of the connections to multiple virtual instruments.

The HiSLIP protocol enables secure encrypted connections. This capability allows the client to verify a server certificate, and the server to verify the client using one of various mechanisms. The encrypted connection may be switched off when transmitting large amounts of non-sensitive data.

1.2 References

Several other documents and specifications are related to this specification. These other related documents are the following:

VPP-4.3	The VISA library Specification defines the client side API. This specification constrains the implementation of the protocol. This specification is available from the IVI Foundation web site at www.ivifoundation.org.
LXI HiSLIP	The LXI HiSLIP Extended Function specifies the server side implementation. This specification constrains the implementation of the protocol. This specification is available from the LXI Consortium web site at www.lxistandard.org.
VXI-11.1, 11.2, and 11.3	These standards define the VXI-11 protocol which is the primary predecessor to HiSLIP.
IEEE 488.2	IEEE Std 488.2 defines the interrupted protocol requirements as well as the appropriate server behavior for several of the GPIB messages.
RFC 4422	Specifies the Simple Authentication and Security Layer (SASL) used for authentication

[1] The IANA assigned port for HiSLIP is 4880

RFC 8446	Specifies the Transport Layer Security (TLS) protocol used for encryption

1.3 Definitions of Terms and Acronyms

This section defines terms and acronyms that are specific to the HiSLIP protocol:

RMT	From IEEE Std 488.2: Response Message Terminator. RMT is the new-line accompanied by END sent from the server to the client at the end of a response. Note that with HiSLIP this is implied by the *DataEND* message.
END	From IEEE Std 488.2: END is a protocol generated indication of the end of a message. It is not indicated with an 8-bit value in the data stream. This message is provided by HiSLIP.
eom	From IEEE Std 488.2: end-of-message. eom is the termination character of a message to the server. The eom is either: new-line, END, or a new-line accompanied by an END. For the purposes of HiSLIP, eom is implicit after group execute trigger.
interrupted	From IEEE Std 488.2: A protocol error indicating that a server received an input message (either a command or query) before the client has fully accepted the response of the preceding message.
HiSLIP	High Speed LAN Instrument Protocol defined in this specification.
MAV	From IEEE Std 488.2: A bit indicating that there is a message available from the server.
SASL	Simple Authentication and Security Layer
TLS	Transport Layer Security

2 HiSLIP Protocol Description

Both the synchronous and asynchronous channels send all command and data information in a fixed packet format. A complete packet is referred to as a message.

2.1 Protocol Version

The HiSLIP protocol version defined in this document is 2.0. The document revision may be different from the protocol version.

2.2 *HiSLIP Port Assignment*

By default, all HiSLIP clients and servers shall use the IANA assigned port number of 4880.

This does not preclude HiSLIP clients and servers configuring to use the HiSLIP protocol on other ports.

2.3 *Message Format*

The messages consist of a header followed by a counted payload. However, the payload count is frequently zero.

Table 2 HiSLIP Message Header Format

Field	Octets	Field Offset
Prologue (ASCII "HS")	2	0
Message Type	1	2
Control Code	1	3
Message Parameter	4	4
Payload Length	8	8
Data	Payload Length	16

Table 2 defines the header used for HiSLIP messages. The fields are:

Prologue — A pattern to facilitate HiSLIP devices detecting when they receive an ill-formed message or are out of sync. The value shall be ASCII 'HS' encoded as a 16 bit value. 'H' is in the most significant network order position and 'S' is in the second byte.

Message Type — This field identifies this message. See Table 3, *HiSLIP Messages*, for a description of the HiSLIP messages. See Table 4,*Message Type Value Definitions,* for the numeric values of each message type.

Control Code — This 8-bit field is a general parameter for the message. If the field is not defined for a message, 0 shall be sent.

Message Parameter — This 32-bit field has various uses in different messages. If this field is not defined for a message, 0 shall be sent.

Payload Length — This field indicates the length in octets of the payload data contained in the message. This field is an unsigned 64-bit integer. The maximum data transfer size may be

limited by the implementation, see section 6.10, *Maximum Message Size Transaction* for details. If the message type does not use a payload, the length shall be set to zero.

All HiSLIP fields are marshaled onto the network in network order (big endian), most significant byte first.
Where the specification calls for an ASCII string as the payload the payload length shall refer to the length of the number of characters in the string and extended ASCII (8-bit) is implied. A trailing NUL character shall not be sent or accounted for in the length.

2.4 Summary of HiSLIP Messages

Table 3 summarizes the HiSLIP messages.

Table 3 HiSLIP Messages

Sender	Channel	Message Type (1 byte)	Control Code (1 byte)	Message Parameter (4 bytes)	Payload (counted field)
C	S	*Initialize*		UpperWord : Client protocol version LowerWord : Client-vendorID	sub-address in ASCII, may be of zero length
S	S	*InitializeResponse*	Bit 0: 0 Prefer Synchronized Bit 0 :1 Prefer Overlap If negotiated protocol version >= 2.0: Bit 1: 0 encryption optional or secure connection capability not supported Bit 1: 1 encryption mandatory Bit 2: 0 Establish Secure Connection Transaction not required after Initialization Transaction Bit 2: 1 Establish Secure Connection Transaction must follow Initialization Transaction. Bit 3..5: reserved for future IVI features Bit 6..7: vendor specific	UpperWord : Negotiated protocol version LowerWord : SessionID	If negotiated protocol version >= 2.0: The payload is reserved
E	E	*FatalError*	ErrorCode (see Table 14)	--	Error Message in ASCII, may be of zero length
E	E	*Error*	ErrorCode (see Table 16)	--	Error Message in ASCII, may be of zero length
C	S	*Data*	Bit 0 : 0 RMT was not delivered Bit 0 : 1 RMT was delivered	MessageID (identifier for this message)	Counted data
S	S		--	MessageID – usage depends on overlapped or synchronized mode	Counted data
C	S	*DataEND*	Bit 0 : 0 RMT was not delivered Bit 0 : 1 RMT was delivered	MessageID (identifier for this message)	Counted data

S	S		--	MessageID – usage depends on overlapped or synchronized mode	Counted data
C	A	*AsyncLock*	1 – Request	Timeout (in milliseconds)	LockString in ASCII, may be of zero length.
			0 – Release	MessageID of last sent message	--
S	A	*AsyncLockResponse*	*In response to request*: 0 – Failure 1 – Success 3 - Error	--	--
			In response to release: 1 - Success exclusive 2 - Success shared 3 - Error	--	--
C	A	*AsyncLockInfo*	--	--	--
S	A	*AsyncLockInfoResponse*	0 – No exclusive lock granted 1 – Exclusive lock granted	Number of HiSLIP clients holding locks when *AsyncLockInfo* was processed	--
C	A	*AsyncRemoteLocalControl*	0 – Disable remote 1 – Enable remote 2 – Disable remote and go to local 3 – Enable Remote and go to remote 4 – Enable remote and lock out local 5 – Enable remote, go to remote, and set local lockout 6 – go to local without changing REN or lockout state	MessageID of last sent message	--
S	A	*AsyncRemoteLocalResponse*	--	--	--
C	A	*AsyncDeviceClear*	--	--	--
S	A	*AsyncDeviceClearAcknowledge*	Feature-bitmap	--	If negotiated protocol version >= 2.0: The payload is reserved
C	S	*DeviceClearComplete*	Feature-bitmap	--	If negotiated protocol version >= 2.0: The payload is reserved

S	S	*DeviceClearAcknowledge*	Feature-bitmap	--	If negotiated protocol version >= 2.0: The payload is reserved
C	S	*Trigger*	0 – RMT was not delivered 1 – RMT was delivered	MessageID (this message)	--
S	S	*Interrupted*	--	MessageID	--
S	A	*AsyncInterrupted*	--	MessageID	
C	A	*AsyncMaximumMessageSize*	--	--	8-byte size – note that the payload length is always 8 and the count is in the payload.
S	A	*AsyncMaximumMessageSizeResponse*	--	--	8-byte size – note that the payload length is always 8 and the count is in the payload.
C	E	*GetDescriptors*			
S	E	*GetDescriptorsResponse*			Counted data. All descriptors returned at once
C	A	*AsyncInitialize*	--	SessionID	--
S	A	*AsyncInitializeResponse*	If negotiated protocol version >= 2.0: Bit 0: 0 secure connection capability is not supported Bit 0: 1 secure connection capability is supported Bit 2..5: 0 (reserved for future IVI features) Bit 6..7: vendor specific	Server-vendorID	If negotiated protocol version >= 2.0: The payload is reserved
S	A	*AsyncServiceRequest*	Server status	--	--
C	A	*AsyncStatusQuery*	0 – RMT was not delivered 1 – RMT was delivered	MessageID of last sent message	--
S	A	*AsyncStatusResponse*	Server status response	--	--
C	S	*StartTLS*			
C	A	*AsyncStartTLS*	0 – RMT was not delivered 1 – RMT was delivered	MessageID of last sent message	4-byte size - MessageID of last received message in network byte order
S	A	*AsyncStartTLSResponse*	0 – busy 1 – success 3 – error		
C	S	*EndTLS*			
C	A	*AsyncEndTLS*	0 – RMT was not delivered 1 – RMT was delivered	MessageID of last sent message	4-byte size - MessageID of last received message in network byte order

S	A	*AsyncEndTLSResponse*	0 – busy 1 – success 3 – error		
C	S	*GetSaslMechanismList*			
S	S	*GetSaslMechanismListResponse*			Counted data containing space separated list of SASL mechanisms
C	S	*AuthenticationStart*			Counted data containing selected mechanism
E	S	*AuthenticationExchange*			Counted data
S	S	*AuthenticationResult*	Bit 0: 0 Authentication failed Bit 0: 1 Authentication successful	If authentication fails, mechanism dependent error code	Counted data may contain additional data if authentication was successful. If authentication was not successful counted data contains human readable error message
E	E	*VendorSpecific*	Arbitrary	Arbitrary	Data

In Table 3 :

In the Sender column :

S indicates	Server generated message
C indicates	Client generated message
E indicates	A message that may be generated by either the client or server

In the channel column :

S indicates	Synchronous channel message
A indicates	Asynchronous channel message
E indicates	A message that may be sent on either the synchronous or asynchronous channel

The following messages carry the RMT-delivered flag: *Data*, *DataEND*, *Trigger*, *AsyncStatusQuery*, *AsyncStartTLS*, and *AsyncEndTLS*.

2.5 Numeric Values of Message Type codes

The Table 4 defines the numeric values of the Message Type codes. Furthermore, the table associates a message type with a capability.

Table 4 Message Type Value Definitions

Message Type	Channel	Numeric Value (decimal)	Server Capability	Minimum HiSLIP protocol version
Initialize	Synchronous	0	General	1.0
InitializeResponse	Synchronous	1	General	1.0
FatalError	Either	2	General	1.0
Error	Either	3	General	1.0
AsyncLock	Asynchronous	4	General	1.0
AsyncLockResponse	Asynchronous	5	General	1.0
Data	Synchronous	6	General	1.0
DataEnd	Synchronous	7	General	1.0
DeviceClearComplete	Synchronous	8	General	1.0
DeviceClearAcknowledge	Asynchronous	9	General	1.0
AsyncRemoteLocalControl	Asynchronous	10	General	1.0
AsyncRemoteLocalResponse	Asynchronous	11	General	1.0
Trigger	Synchronous	12	General	1.0
Interrupted	Synchronous	13	General	1.0
AsyncInterrupted	Asynchronous	14	General	1.0
AsyncMaximumMessageSize	Asynchronous	15	General	1.0
AsyncMaximumMessageSizeResponse	Asynchronous	16	General	1.0
AsyncInitialize	Asynchronous	17	General	1.0
AsyncInitializeResponse	Asynchronous	18	General	1.0
AsyncDeviceClear	Asynchronous	19	General	1.0
AsyncServiceRequest	Asynchronous	20	General	1.0
AsyncStatusQuery	Asynchronous	21	General	1.0
AsyncStatusResponse	Asynchronous	22	General	1.0
AsyncDeviceClearAcknowledge	Asynchronous	23	General	1.0
AsyncLockInfo	Asynchronous	24	General	1.0
AsyncLockInfoResponse	Asynchronous	25	General	1.0
GetDescriptors	Either	26	General	2.0
GetDescriptorsResponse	Either	27	General	2.0
StartTLS	Synchronous	28	SC	2.0
AsyncStartTLS	Asynchronous	29	SC	2.0
AsyncStartTLSResponse	Asynchronous	30	SC	2.0
EndTLS	Synchronous	31	SC	2.0
AsyncEndTLS	Asynchronous	32	SC	2.0
AsyncEndTLSResponse	Asynchronous	33	SC	2.0
GetSaslMechanismList	Synchronous	34	SC	2.0
GetSaslMechanismListResponse	Synchronous	35	SC	2.0
AuthenticationStart	Synchronous	36	SC	2.0
AuthenticationExchange	Synchronous	37	SC	2.0
AuthenticationResult	Synchronous	38	SC	2.0
reserved for future use		39-127	Not defined	
VendorSpecific	Either	128-255 inclusive	Not defined	

In Table 4 in the “Server Capability” column:

- “General” indicates the message type is supported by all HiSLIP servers and clients.
- “SC” indicates the message type can be used if the server indicates support for the “Secure connection” capability in the *AsyncInitializeResponse* response.
- “Not defined” means that the associated capability of a reserved or vendor specific message type is not defined.

In Table 4 the “Minimum HiSLIP protocol version” column indicates the minimum HiSLIP version since which a message type is supported. If the negotiated protocol version is below this value, the server does not support the given message type and the client must not send messages of this type.

2.6 Locking Mechanism

The HiSLIP protocol provides a locking mechanism including exclusive and shared locks.

After the Initialization Transaction, all subsequent HiSLIP messages from the client to the server are subject to access control governed by the Lock Transaction.

Section 6.5 describes how one or more clients can obtain a lock. If the server is locked, only the client or clients that have been granted the lock are permitted access to the server. Clients that do not have the lock behave as described in section 2.6.1.

If a client is holding an exclusive lock, the server guarantees that only that client has received an exclusive lock. If no client is holding an exclusive lock, then multiple clients may be granted a shared lock. If multiple clients hold a shared lock, each is permitted access to the server. If the server has no outstanding locks, then any client is permitted access to the server.

If any client is holding a shared lock, only a client holding a shared lock shall be granted an exclusive lock. However, if no client is holding a shared lock, a client requesting an exclusive lock shall be granted that lock.

If a server receives a *Lock* message requesting a lock from a client that is not holding the lock, it will wait for the timeout time indicated in that message for the lock to become available.

If a HiSLIP connection is closed any locks assigned to the corresponding client are immediately released by the server.

Nothing in this section should be taken to require that a server is precluded from implementing other security mechanisms that may result in it refusing to grant access to any client.

The behavior of a server that supports connections other than HiSLIP is beyond the scope of this specification. However, it is appropriate for a server to manage a single lock across several connection styles, and therefore, a HiSLIP client may be refused a lock although no other HiSLIP client is holding a lock. Also, a HiSLIP lock may impact other connection styles.

2.6.1 Server Behavior When Client Does Not Have a Lock

This section describes how servers behave towards clients that do not have the lock when some other client has been granted either an exclusive or shared lock.

Exchanges with a client that does not have a lock while another client has been granted a lock are handled as follows:

- The *AsyncDeviceClear* transaction (including all protocol messages) is completed immediately per section 6.12,*Device Clear Transaction*. Note that only the channel upon which the *AsyncDeviceClear* was sent is impacted by this transaction. In this circumstance, device clear is not permitted to impact any of the server functions other than those associated with this HiSLIP communication session.
- The Establish Secure Connection Transaction and the End Secure Connection Transaction (including all protocol messages) are completed immediately per sections 6.15 and 6.16, respectively.
- Any transactions that are in-process are completed normally. Thus, the following may be sent:
    - *AsyncLockResponse*
    - *AsyncInterrupted*
    - *AsyncMaximumMessageSizeResponse*
    - Appropriate *VendorSpecific* messages (subject to the vendor-specific definition of these messages)

- *AsyncStatusQuery*, *AsyncMaximumMessageSize* and *AsyncLockInfo* transactions are completed.
- *AsyncRemoteLocalControl* transactions are completed. However, the client shall not wait for locks on other clients to be released. If another client is holding a lock, the server shall only act on the remote/local request after the lock is released. The behavior is device dependent.
- *AsyncServiceRequest* messages are sent normally.
- All synchronous messages are left in the input buffer. Normal TCP behavior to prevent buffer overflows takes place. *Data*, *DataEND*, *Trigger*, *Error*, *FatalError*, and *VendorSpecific* messages are left in the input buffer.
- Data traffic from the server to the client (*Data* and *DataEND* messages) are sent normally. However, since incoming synchronous messages are blocked, only responses to operations that were begun before the lock was granted will be generated.

2.7 HiSLIP Buffer Sizes

HiSLIP provides the *AsyncMaximumMessageSize* transaction to inform the client and server of the largest message they are permitted to send to the other device on the synchronous channel. Clients should initiate this transaction as part of the initialization process (shortly after the HiSLIP initialization) in order to guarantee that messages do not overflow the other devices buffers.

Each device shall also have a buffer suitable to accept any asynchronous message. The buffer size for asynchronous messages are dominated by the messages with a variable payload. That is:

- *Error* and *FatalError* which include a variable length string in addition to the 16 byte message header
- *AsyncLock* message (sent to the server) that includes the variable length lock string.

Common client implementations limit the payload size to 256 bytes.

If either the server or client detect the arrival of a message that is too large to properly handle it shall send an *Error* message with the appropriate HiSLIP defined error value.

3 Overlapped and Synchronized Modes

In order to maintain compatibility with GPIB, VXI-11 and USB-TMC instruments, the HiSLIP protocol supports two different operating modes:

Overlap mode	In overlap mode input and output data and trigger messages are arbitrarily buffered between the client and server. For instance, a series of independent query messages can be sent to the server without regard to when they complete. The responses from each will be returned in the order the queries were sent. Thus multiple query operations may be initiated and conducted by the server independent of the rate at which the client consumes the responses.
Synchronized mode	In Synchronized mode, the client is required to read the result of each query message before sending another query[2]. If the client fails to do so, the protocol generates the interrupted protocol error and the response from the preceding query is cleared by the protocol.

All HiSLIP clients shall support both synchronized and overlapped mode. HiSLIP servers shall support either synchronized or overlapped mode or both.

The following sections describe the implementation of these modes.

Note that the calculation of message available (MAV) and the *AsyncStatusQuery* transaction differ between the overlapped and synchronized modes also. See section 6.14, *Status Query Transaction* for details.

[2] Per the IEEE 488.2 definition of a response message.

3.1 Synchronized Mode

Synchronized mode closely mimics the requirements of the IEEE Std 488.2 message exchange protocol to detect the interrupted error.

3.1.1 Synchronized Mode Server Requirements

HiSLIP servers shall implement the following:

1. When the server application layer (nominally an instrument parser) requests HiSLIP to send a response message terminator, the server shall verify that no data is in the server input queue. If there is data in the input queue, the server shall declare an interrupted error.

   To declare an interrupted error, the server shall:

   - Use the server error reporting mechanism to report the interrupted error within the server.
   - Clear the response message just received from the server application layer, and any other messages buffered to be sent to the client.
   - Send the *Interrupted* transaction to the client, including both the *Interrupted* and *AsyncInterrupted* messages.

2. When receiving an RMT-delivered flag-carrying message verify the state of the RMT-delivered flag.

   The server shall maintain a flag that indicates RMT-expected. RMT-expected shall be set *true* when the server sends a *DataEND* message (that is, when it sends an RMT).

   The RMT-expected bit shall be cleared when the server receives *AsyncStatusQuery*, *AsyncStartTLS*, or *AsyncEndTLS* with RMT-delivered flag set to true.

   When *Data, DataEND,* or *Trigger* are received, if RMT-expected and RMT-delivered are both either true or false the RMT-expected bit shall be cleared.

   When *Data, DataEND,* or *Trigger* are received, if RMT-expected and RMT-delivered are different, the server shall declare an interrupted error. The server shall use the server error reporting mechanism to report the interrupted error. No indication of this interrupted error is sent to the client by the HiSLIP protocol.

When servers send the *DataEND* message, they shall set the MessageID field to the MessageID of the client message that contained the eom that generated this response.

When servers send the *Data* message, they may set the MessageID field to the MessageID of the client message that contained the eom that generated this response so long as that eom is at the end of the identified message. In some circumstances (for instance, if the eom is not at the end of the message), the server might not be able to provide the MessageID of the message ending in the eom. In these circumstances, the server shall set the MessageID to 0xffff ffff.

After interrupted error processing is complete, the server resumes normal operation.

3.1.2 Synchronized Mode Client Requirements

HiSLIP clients shall implement the following:

1. When receiving *DataEND* (that is an RMT), verify that the MessageID indicated in the *DataEND* message is the MessageID that the client sent to the server with the most recent *Data, DataEND* or *Trigger* message.

   If the MessageIDs do not match, the client shall clear any *Data* responses already buffered and discard the offending *DataEND* message.

2. When receiving *Data* messages if the MessageID is not 0xffff ffff, then verify that the MessageID indicated in the *Data* message is the MessageID that the client sent to the server with the most recent *Data*, *DataEND* or *Trigger* message.

   If the MessageIDs do not match, the client shall clear any *Data* responses already buffered and discard the offending *Data* message.

3. When the client sends *Data*, *DataEND* or *Trigger*, if there are any whole or partial server messages that have been validated per rules 1 and 2 and buffered, they shall be cleared.

4. When the client receives *Interrupted* or *AsyncInterrupted*, it shall clear any whole or partial server messages that have been validated per rules 1 and 2.

   If the client initially detects *AsyncInterrupted*, it shall also discard any further *Data* or *DataEND* messages from the server until *Interrupted* is encountered.

   If the client detects *Interrupted* before it detects *AsyncInterrupted*, the client shall not send any further messages until *AsyncInterrupted* is received.

Clients shall maintain a MessageID count that is initially set to 0xffff ff00. When clients send *Data, DataEND* or *Trigger* messages, they shall set the message parameter field of the message header to the current MessageID and increment the MessageID by two in an unsigned 32-bit sense (permitting wrap-around).

The MesssageID is reset to 0xffff ff00 after device clear, and when the connection is initialized.

After interrupted error processing is complete, the client resumes normal operation.

3.2 Overlapped mode

In overlapped mode commands and responses are buffered by the client and server and I/O operations are permitted to overlap.

No special processing is required in the server or client other than buffering inbound messages until the respective application layer requires them. Buffers are only cleared by a device clear.

3.2.1 Overlap Mode Server Requirements

HiSLIP overlap mode servers maintain a MessageID and use it as follows:

1. The MessageID shall be reset to 0xffff ff00 after device clear or initialization.

2. When the server sends *Data* or *DataEND* messages it shall place the MessageID into the message parameter and increment it by two in an unsigned 32-bit fashion (permitting wrap-around).

3.2.2 Overlap Mode Client Requirements

HiSLIP clients shall implement the following:

1. In overlap mode, when sending *AsyncStatusQuery*, the client shall place the MessageID of the most recent message that has been entirely delivered to the client in the message parameter.

Clients shall maintain a MessageID count that is initially set to 0xffff ff00. When clients send *Data, DataEND* or *Trigger* messages, they shall set the MessageID field of the message header to the current MessageID and increment the MessageID by two in an unsigned 32-bit sense (permitting wrap-around).

The MesssageID is reset after device clear, and when the connection is initialized. In overlap mode, the MessageID is only used for locking.

4 Server Capabilities

Server capabilities are added in HiSLIP version 2.0 to enable extensions to the HiSLIP protocol. A server announces server capabilities in the *AsyncInitializeResponse* message. Table 5 *Server Capabilities*, lists defined server capabilities. Table 4 *Message Type Value Definitions*, documents message types that are general and message types that are unique to a server capability. Adding a new capability increases the revision number of this document, but does not increase the HiSLIP protocol version. Server vendors may define vendor specific capabilities. The following sections describe each capability in detail.

Table 5 Server Capabilities

Server Capability Name	Description
Secure Connection	Support of encrypted connections and authentication.

4.1 Secure Connection

HiSLIP protocol version 2.0 enables secure connections. A secure connection is cryptographically protected against attackers trying to read or manipulate the transferred data. This is achieved using the Transport Layer Security (TLS) for encryption and decryption. HiSLIP version 2.0 also enables authentication: the server authenticates its identity by sending an X.509 certificate to the client when the TLS connection is established. The client authenticates to the server using a server supported SASL (Simple Authentication and Security Layer) mechanism described below.

The requirements in section 4.1, *Secure Connection*, only apply to servers supporting the Secure Connection capability.

4.1.1 Client authentication mechanisms

The client may use any mechanism supported by the server, which may support any authentication mechanisms defined by the SASL, including:

- ANONYMOUS: the client sends an anonymous token string
- EXTERNAL: the server authenticates a client provided certificate
- GSSAPI: login with a Kerberos ticket
- NTLM: Microsoft Windows login
- PLAIN: username and password login
- others (cf. RFC 4422)

The server shall support at least one SASL authentication mechanism.

The server checks the credentials provided by the client and either grants or denies access. The means of checking the validity of the client's credentials is out of scope of this specification.

If the server check of client provided information results in an authentication failure, the client may try authentication using a different mechanism.

4.1.2 Server certificate

The server shall provide a valid X.509 v3 certificate to the client when the TLS connection is established. This certificate must be signed by a certificate authority trusted by the client. The device vendor shall ensure that the device (HiSLIP server) is equipped with a suitable X.509 certificate. This specification does not specify how the certificate is deployed to a device or what the contents of certificate's fields are.

This paragraph is for informative purposes only. A device vendor has options for deploying certificates to the device. To maximize interoperability, other specifications might put additional constraints on the certificate. For example, a device vendor may install an LXI IDevID during manufacturing. As an alternative, a device vendor could also install vendor-signed certificates, allow the device to create self-signed certificates, or allow the device owner to install a certificate.

4.1.3 Encryption modes

For encryption, HiSLIP supports two modes of operation:

- encryption mandatory
- encryption optional

The server is configured to operate in one of the two modes. The server announces the mode in the control code of the *InitializeResponse* message. The encryption optional mode is intended for backward compatibility as it provides a way for HiSLIP protocol version 1.0 clients to communicate with HiSLIP protocol version 2.0 servers. This mode also lets the client optimize performance on demand. Details of both encryption modes are given in Table 6 *Encryption modes*.

Table 6 Encryption modes

Affected HiSLIP item	encryption mandatory	encryption optional
Negotiated protocol version	Negotiated protocol version must be >= 2.0.	Any protocol version.
Initialization	Establish Secure Connection Transaction shall follow Initialization Transaction. Only Maximum Message Size Transaction is allowed in between.	After Initialization Transaction the Establish Secure Connection Transaction may be performed. If the server requires authentication when establishing connection, it shall be performed.
TLS Encryption	Client cannot switch encryption off. If client switches encryption off, server closes the connection.	Client may toggle between encryption on and off by using the Establish Secure Connection Transaction and End Secure Connection Transaction.
Authentication	Authentication valid for entire lifetime of connection	Authentication only valid on encrypted connections and until encryption is switched off

4.1.4 Initial Encryption Modes

If encryption is optional the server may require that the client switches encryption on when establishing the connection. This is useful if the owner of a device wants to restrict access to selected users and operates the device in a secure network. Hence authentication is required, but the encryption mode is optional.

The server announces the mode in the initial encrypted connection required bit (bit 2) of the control code of the *InitializeResponse* message. The value of this bit is only relevant if the encryption mode it optional. If the server announces encryption mode mandatory, it shall also announce initial encrypted connection required. If the initial encrypted connection required bit is set, the client shall perform the Establish Secure Connection Transaction after the Initialization Transaction (with only Maximum Message Size Transaction allowed in between). If the client fails to do so, the server shall announce fatal error 5 and close the connection. If the initial encrypted connection required bit is not set, the client may perform the Establish Secure Connection Transaction at its discretion.

5 Descriptors

Descriptors are added in HiSLIP version 2.0 to provide extra information about specific server capabilities.

All descriptors shall follow the format below.

Table 7 Descriptor format

Offset	Field	Size in bytes	Content/Value	Description
0	Length	2	Number	Size of the descriptor content beginning at offset 3, in bytes Size = *N*
2	Descriptor type	1	Byte	
3	Varies	*N* Varies	Varies	Varies. Content depends on descriptor type. Content may include many instances of different data types including bitfields, bytes, Uint16's, Uint32's, Uint64's, strings, etc.

5.1 Descriptor types

The numeric values of the different descriptor types are given in Table 8 *Numeric values of descriptor types*.

Table 8 Numeric values of descriptor types

Descriptor Type (1 byte)	Meaning
0	Supported TLS versions descriptor
1	TLS information descriptor
2	TLS last error descriptor
3 - 127	IVI reserved
128 - 255	Vendor Specific

5.1.1 Supported TLS versions descriptor

Table 9 Supported TLS versions descriptor

Offset	Field	Size in bytes	Content/Value	Description
0	Length	2	Number = *N*	Size of the descriptor content beginning at offset 3, in bytes
2	Descriptor type	1	Byte = 0	This is the Supported TLS versions descriptor.
3	Array of supported TLS versions encoded according to TLS standard in network byte order (RFC 8446 section 5.1)	*N* = 2*(number of supported TLS versions)	Array of UInt16	Example 0x0303 // TLS 1.2 is supported 0x0304 // TLS 1.3 is supported

5.1.2 TLS information descriptor

Table 10 TLS information descriptor

Offset	Field	Size in bytes	Content/Value	Description
0	Length	2	Number = *N*	Size of the descriptor content beginning at offset 3, in bytes
2	Descriptor type	1	Byte = 1	This is the TLS information descriptor.
3	TLS information string	*N* = length of string.	ASCII-7 String Not terminated with 0x00	Examples • "`TLS 1.2, TLS 1.3, ECDHE disabled`" • "`TLS 1.3; GSASL 1.8;IDEVID:LXI`"

5.1.3 TLS last error descriptor

Table 11 TLS last error descriptor

Offset	Field	Size in bytes	Content/Value	Description
0	Length	2	Number = *N*	Size of the descriptor content beginning at offset 3, in bytes
2	Descriptor type	1	Byte = 2	This is the TLS last error descriptor.
3	TLS last error string	*N* = length of string.	ASCII-7 String Not terminated with 0x00	Example: "`SSL_accept() FAILED; sslErr: SSL_ERROR_SSL; A failure in the SSL library occurred, usually a protocol error. The OpenSSL error queue contains more information on the error`"

6 HiSLIP Transactions

The following sections describe the HiSLIP protocol transactions.

In the following sections angle brackets (<>) are used to separate the various fields of the message. This is always expressed as four fields, the fourth field represents the payload and a count is always implied. <0> in the payload field indicates a count of zero and no payload.

Some transactions are only available if the server supports a certain capability. In this case the associated capability is stated in the descriptive text of the transaction.

6.1 Initialization Transaction

The purpose of the Initialization Transaction is to establish the HiSLIP connection between the client and the server. This requires opening a synchronous and an asynchronous channel on the same server port and associating the two together. The two are associated through a session ID that is provided to the client by the server in response to the *Initialize* message.

Servers shall support multiple simultaneous clients initializing.

Table 12 Initialization Transaction

Step	Initiator	Message content	Action
0	Server	none	Server passively opens TCP server socket on the IANA assigned port.
1	Client	Opens the synchronous TCP connection (TCP SYN message)	Client does an active TCP open, the server continues to wait for additional connections
2	Client	<*Initialize*><0><*upper:*client-protocol-version : *lower :*client-vendorID><sub-address>	Client starts the initialization by identifying the vendor, specifying the sub-address, and advertising the protocol version it supports.
3	Server	<*InitializeResponse*><*bit 0*: overlap-mode; *bit 1*: encryption-mode; *bit 2*: initial encryption><*upper :*negotiated-protocol-version : *lower* :SessionID><0>	Server responds with negotiated protocol version, which is the lower of the server version and the client version. Server announces whether encryption is mandatory or optional and whether it expects that this transaction is followed by the Establish Secure Connection Transaction. The server also provides the SessionID to send with *AsyncInitialize*. Note that several *FatalError* messages are appropriate at this time.
4	Client	Opens the asynchronous TCP connection (TCP SYN message)	Client opens second connection for the asynchronous channel on same server port
5	Client	<*AsyncInitialize*><*0*><SessionID><0>	The client sends SessionID to associate this TCP session with the established HiSLIP synchronous channel.
6	Server	<*AsyncInitializeResponse*><server-capabilites><server-vendorID><0>	Server acknowledges initialize and provides the vendor ID. Server announces supported capabilities. If initial encryption is not required or mandatory the HiSLIP connection is ready for use. Otherwise, proceed with Establish Secure Connection Transaction.

The following are the fields in the *Initialize* client message:

client-protocol-version This identifies the highest version of the HiSLIP specification that the client implements. Per the IVI standards requirements, HiSLIP specification versions are of the form <major>.<minor>. The major specification revision, expressed as a binary 8-

bit integer is the first byte of the client version. The minor number expressed as a binary 8-bit integer is the second byte of the client version.

The client version is sent in the most significant 16-bits (big endian sense) of the 32-bit message parameter.

client-vendorID — This identifies the vendor of the HiSLIP protocol on the client. This is the two-character vendor abbreviation from the VXI*plug&play* specification VPP-9. These abbreviations are assigned free of charge by the IVI Foundation[3].

The client vendorID is sent in the least significant 16-bits (big endian sense) of the 32-bit message parameter.

sub-address — It identifies a particular device managed by this server. It is in the payload field and therefore includes a 64-bit count. The count is followed by the appropriate length ASCII sub-address. For instance: “device2”. The maximum length for this field is 256 characters.

If the sub-address is null (zero length) the initialize opens the default (perhaps only) device at this IP address.

For VISA clients this field corresponds to the VISA LAN device name. Note that VISA requires that such HiSLIP device names begin with ‘hislip’ and contain only alphanumeric characters, with a default device name of ‘hislip0’.

The following are the fields in the *InitializeResponse* server message:

server-protocol-version — This identifies the highest version of the HiSLIP specification that the server implements. It is expressed the same as the client-version field in the *Initialize* client message.

The server version is sent in the most significant 16-bits (big endian sense) of the 32-bit message parameter.

SessionID — This is used to associate the synchronous and asynchronous connections and must be provided by the client in the *InitializeAsync* message. This associates the two TCP connections into a single HiSLIP connection.

The client vendorID is sent in the least significant 16-bits (big endian sense) of the 32-bit message parameter.

overlap-mode — The server uses this field to indicate if it is initially in overlapped or synchronous mode. 0 indicates synchronous, 1 indicates overlapped.

encryption-mode — The server uses this field to indicate if encryption is optional or mandatory. If the server does not support the Secure Connections Capability this bit shall be set to 0.

initial encryption — The server uses this field to indicate if the client shall switch to the encrypted mode when establishing the connection. This bit may only be set if the encryption-mode bit is not set.

[3] Contact the IVI Foundation (admin@ivifoundation.org) to register a new vendor ID (also known as a vendor prefix). Vendors do not need to join the IVI Foundation to obtain a defined two-character abbreviation.

The following is the field in the *AsyncInitialize* message:

SessionID	This is the session ID provided by the server in the *InitializeResponse* message. It associates the synchronous and asynchronous connections. This may be discarded by the client after this message.

The following is the field in the *AsyncInitializeResponse* message:

server-capabilities	Announcement of capabilities supported by the server. If bit 0 is set the secure connection capability is supported.
server-vendorID	This identifies the vendor of the server. This is the two-character vendor abbreviation from the VXI*plug&play* specification VPP-9. These abbreviations are assigned free of charge by the IVI Foundation.[4]

After the initialization sequence, the client and server will both use the highest protocol version supported by both devices (that is, the smaller of the two exchanged versions). Note that all HiSLIP devices must support earlier protocol versions.

Clients that require exclusive access to the server must immediately follow the Initialization Transaction with an appropriate *Lock* transaction. If the *Lock* operation fails then the client can close the connection. Servers shall not automatically grant a lock to new clients, and the connection may be opened when another client is holding the lock. If the server announces the mandatory encryption mode or initial encryption is required, the *Lock* transaction must immediately follow the Establish Secure Connection Transaction instead of the initialize transaction.

If the server announces that encryption is mandatory or that initial encryption is required and the client does not proceed with either the Establish Secure Connection Transaction or Maximum Message Size Transaction, the server declares the fatal error 5 (Secure connection failed).

If the client closes the connection after receiving the *InitializeResponse*, the server should not declare an error as this is a legitimate way for a client to validate the presence and version of a server.

[4] Ibid

6.2 Fatal Error Detection and Synchronization Recovery

Table 13 Synchronous Fatal Error Message

Initiator	Message	Data Consumer
Either client or server	<*FatalError*><ErrorCode><0><length><message>	Accept data and handle appropriately
Initiator	Close the connection	If initiator is the client, it may re-open the connection per 6.1

At any point, the client or server may encounter a non-recoverable error situation. For instance, the prologue may be incorrect. If either device detects an error condition that is likely to cause the two devices to lose synchronization it shall send the *FatalError* message on the synchronous channel and the asynchronous channel with appropriate diagnostic information.

The IVI Foundation defines the error codes listed in Table 14. Error codes from 128-255 inclusive are device defined.

The payload shall be of the specified length and contain a human readable error description expressed in ASCII. A length of zero with no description is legal.

If the error is detected by the client, after sending the *FatalError* messages it shall close the HiSLIP connection and may attempt to re-establish the connection (that is, close both synchronous and asynchronous connections and re-establish the connection per section 6.1, *Initialization Transaction*).

If the error is detected by the server, after sending the *FatalError* messages, it shall close the HiSLIP connection. The client may re-establish the connection. However the SessionID for the new session will not necessarily relate to the previous SessionID. Note that locks will not be retained and must be re-acquired.

Table 14 HiSLIP Defined Fatal Error Codes

Error Code	Message
0	Unidentified error
1	Poorly formed message header
2	Attempt to use connection without both channels established
3	Invalid Initialization Sequence
4	Server refused connection due to maximum number of clients exceeded
5	Secure connection failed
6-127	Reserved for HiSLIP extensions
128-255	Device defined errors

6.3 Error Notification Transaction

Table 15 Synchronous Error Notification Transaction

Initiator	Message	Data Consumer
Either client or server	<*Error*><ErrorCode><0><length><message>	Accept data and handle appropriately
Initiator	No further action	No further action

If either the client or server receive a message that it is unable to process but that does not cause it to lose synchronization with the sender it shall discard the errant message and any payload associated with it, then reply with the *Error* message.

The *Error* message shall be sent on whichever connection (synchronous or asynchronous) that the errant message arrived on.

The payload shall be of the specified length and contain a human readable error description expressed in ASCII. A length of zero with no description is legal.

The IVI Foundation defines the error codes listed in Table 16, error codes from 128-255 inclusive are device defined.

After sending the *Error* message, the device shall return to normal processing.

For example, the *Error* message should be sent in reply to unrecognized vendor specific messages or unsupported MessageIDs or control codes.

Table 16 HiSLIP Defined Error Codes (non-fatal)

Error Code	Message
0	Unidentified error
1	Unrecognized Message Type
2	Unrecognized control code
3	Unrecognized Vendor Defined Message
4	Message too large
5	Authentication failed
6-127	Reserved for HiSLIP extensions
128-255	Device defined errors

6.4 DataTransfer Messages

Table 17 Data Transfer Messages from Client to Server

Initiator	Message	Data Consumer
client	<*Data*><RMT-delivered><MessageID><length><data>	Accept data and use it appropriately RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the client delivered RMT to the application layer. The client increments the MessageID with each *Data*, *DataEND* or *Trigger* message sent.
client	<*DataEND*><RMT-delivered><MessageID><length><data>	Accept data and use it appropriately. Final data byte has an accompanying END. RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the client delivered RMT to the application layer. The client increments the MessageID with each *Data*, *DataEND* or *Trigger* message sent.

Table 18 Data Transfer Messages from Server to Client

Initiator	Message	Data Consumer
server	<*Data*><0><MessageID><length><data>	Accept data and handle appropriately The MessageID in synchronized mode is the ID of the message containing the RMT that generated this response or 0xffff ffff. The MessageID in overlapped mode is an ID that is incremented with each data transfer message sent.
server	<*DataEND*><0><MessageID><length><data>	Accept data and handle appropriately. Final data byte has an accompanying END. The MessageID in synchronized mode is the ID of the message containing the RMT that generated this response. The MessageID in overlapped mode is an ID that is incremented with each data transfer message sent.

Either the server or the client is permitted to initiate a data transfer at any time.

For client originated message:

RMT-delivered	RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the HiSLIP client delivered an RMT to the client application layer.
MessageID	MessageID identifies this message so that response data from the server can indicate the message that generated it. For generation of the MessageID, see sections 3.1.2 (*Synchronized Mode Client Requirements*) and 3.2.2 (*Overlap Mode Client Requirements*).

For server originated messages:

MessageID	MessageID in synchronized mode identifies the client message responsible for generating this response. In overlapped mode, the MessageID is a continuously incrementing count that assists in MAV generation. See section 3.1.1(*Synchronized Mode Server Requirements)* and 3.2.1(*Overlap Mode Server Requirements*), and section 6.14.2, *MAV Generation in Overlapped Mode*

The *DataEND* message indicates that the END message should be processed with the final data byte.

These messages are not acknowledged.

6.5 Lock Transaction

Table 19 Lock Transaction – Requesting a Lock

Step	Sender	Message	Action
1	Client	<*AsyncLock*><1=request><timeout><LockString length><LockString>	Request lock, wait up to timeout milliseconds for it to become available. LockString is an ASCII string indicating shared lock identification. A zero length string indicates an exclusive lock request.
2	Server	<*AsyncLockResponse*><0=failure, 1=success, 3=error><0><0>	Response indicates if the lock was successful.

Table 20 Lock Transaction – Releasing a Lock

Step	Sender	Message	Action
1	Client	<*AsyncLock*><0 =release><MessageID><0>	Release lock
2	Server	<*AsyncLockResponse*><1=success exclusive, 2=success shared, 3=error><0><0>	Response indicates the type of lock released if any

The *AsyncLock* client message is used to request or release a lock as described in Table 21.

The following are the fields in the *AsyncLock* transaction:

LockString — an ASCII string that identifies this lock

MessageID — designates the last message to be completed before the release takes place.

Timeout — The *AsyncLock* request passes a 32-bit timeout in the MessageID field. This is the amount of time in milliseconds the client is willing to wait for the lock to grant. If the lock is not available in this amount of time, the *AsyncLockResponse* will fail and return failure. A timeout of 0 indicates that the server should only grant the lock if it is available immediately.

The server shall always reply with an *AsyncLockResponse* per Table 21.

Table 21 Lock request/release operation descriptions

***Lock* Control Code**	***LockResponse* Control Code**	**Description**
0 (release)	1 (success)	Release of exclusive lock was granted.
	2 (success)	Release of shared lock was granted
0 (release)	3 (error)	Invalid attempt to release a lock that was not acquired.
1 (request)	0 (fail)	Lock was requested but not granted (timeout expired)
1 (request)	1 (success)	The lock was requested and granted
1 (request)	3 (error)	Invalid (redundant) request that is, requesting a lock already granted

A null LockString in the *AsyncLock* LockString with a Control Code of 1 indicates a lock request for an exclusive lock.

A non-null LockString in the *AsyncLock* LockString with a Control Code of 1 indicates a request for a shared lock.

An *AsyncLock* with the Control Code set to 0 indicates a request to release the lock.

HiSLIP servers shall respond to lock requests as shown in Table 22. Note that the 'lock state' described in this table is the lock state across all active HiSLIP sessions.

After the Initialization Transaction and Device Clear Transaction, the client shall use the MessageID = 0xfffffefe (0xffffff00-2) in the *AsyncLock* (release) message.

Table 22 Lock Behavior

Initial State	Lock request	Client	New State
Unlocked (initial state)	Shared lock	any client → success	Shared Locked
	Exclusive lock	any client → success	Exclusive Locked
	Release	any client → error	Unlocked
Exclusive locked	Shared Lock	holder of exclusive lock → error	Exclusive Locked
		other client → fails after lock timeout	Exclusive Locked
	Exclusive lock	client not holding the lock → fails after lock timeout	Exclusive Locked
		holder of exclusive lock → error	Exclusive Locked
	Release	holder of exclusive → succeeds	Unlocked
		client without exclusive lock → error	Exclusive locked
Shared locked	Shared lock	holder of shared lock → error	Shared Locked
		other client with right key → succeeds	Shared Locked
		other client with wrong key → fails after lock timeout	Shared Locked
	Exclusive lock	holder of shared lock → succeeds	Both Locks
		client not holding shared lock → fails after lock timeout	Shared Locked
	Release	holder of the shared lock when 2 or more are holding shared lock → succeeds	Shared Locked
		the only remaining holder of the shared lock → Succeeds	Unlocked
		client holding no locks → error	Shared Locked
Both locks	Shared Lock	holder of shared lock → error	Both locks
		other client with right key → succeeds	Both locks
		other client with wrong key → fails after lock timeout	Both locks
	Exclusive lock	client not holding the exclusive lock → fails after timeout	Both locks
		client holding the exclusive lock → error	Both locks
	Release	holder of exclusive and shared lock → succeeds	Shared locked
		other client holding a shared lock → succeeds	Both Locks
		client holding no locks → error	Both locks

HiSLIP servers shall:

1. Go to the Unlocked state when the first connection is initialized
2. Release all locks assigned to a client when that client connection closes

6.5.1 Unlock Considerations

In an unlock operation, the MessageID in the message parameter designates the last *Data, DataEND* or *Trigger* message to be completed before the lock is released. The client is only allowed to specify messages that were transmitted before the *AsyncLock* operation. The *AsyncLock* transaction will not complete the unlock until that message is complete.

Because the *AsyncLockResponse* message sent for an unlock operation is indistinguishable from the same message in response to a lock operation, clients should not initiate a lock operation until the prior unlock *AsyncLockResponse* is received or cancelled via device clear. For the same reason, servers should report an error using the error transaction if they receive an *AsyncLock* message before a pending unlock *AsyncLockResponse* has been sent. While the unlock transaction does not complete until the designated message is complete, the timing of the unlock *AsyncLockResponse* message is left up to the server, as long as the server knows what the eventual unlock transaction outcome will be. As

a result, receipt of the unlock *AsyncLockResponse* message with a success outcome does not mean the server has released the session's lock yet.

The *AsyncLock* unlock operation can only be aborted by device clear. The normal device clear behavior will abandon any pending transactions the unlock operation may be waiting for. Note that this requires the server respond in a timely fashion to an *AsyncDeviceClear* message while waiting for an *AsyncLock* unlock operation to complete. When an *AsyncLock* unlock operation is abandoned by a device clear, the lock shall be released per the pending unlock operation and no confirmation sent to the client.

6.6 Lock Info Transaction

Table 23 Lock Info Transaction

Step	Sender	Message	Action
1	Client	<*AsyncLockInfo*><0><0><0>	Request from the client for lock information.
2	Server	<*AsyncLockInfoResponse*><exclusive-locks-granted><locks-granted><0>	The server returns information regarding locks it has granted.

The following are the fields in the *AsyncLockInfoResponse:*

exclusive-locks-granted 1 if an exclusive lock has been granted and 0 otherwise.

locks-granted the number of clients that were holding locks when *AsyncLockInfo* was processed. A client holding both a shared and exclusive lock is counted only once.

The *LockInfo* transaction is used by the client to determine how many other clients are connected and how many locks have been granted. These values are sampled values from the server and may be inaccurate since other clients may be simultaneous releasing and requesting locks or connections. However, the locks-granted and clients-connected values shall be self-consistent at some point in time when the *LockInfo* was processed by the server.

This transaction is processed regardless of whether the client currently holds a lock.

6.7 Remote Local Transaction

HiSLIP supports GPIB-like remote/local control. The purpose of remote/local is to:

- Prevent front panel input from interfering with remote operations
- Permit front panel local key to re-enable the front panel input
- Provide a way to lockout the local key[5] when the controller needs exclusive access

Table 24 RemoteLocal Control Transaction

Step	Sender	Message	Action
1	Client	<*AsyncRemoteLocalControl*><request><MessageID><0>	Request remote local operation
2	Server	<*AsyncRemoteLocalResponse*><0><0><0>	Confirm remote/local request

request — The values of the request field are shown in Table 25, *Remote Local Control Transactions*

MessageID — designates the MessageID of the most recent *Data*, *DataEND,* or *Trigger* message sent by the client.

The server is permitted to act on the *AsyncRemoteLocalControl* immediately, or wait until after the preceding operations have been acted on by the server.

In some conditions, TCP may deliver the remote/local requests before it delivers a preceding *Data/DataEND* or *Trigger* message generated by the client. The server should consult the MessageID provided with the *AsyncRemoteLocalControl*. If this MessageID is not equal to the MessageID of the last received *Data*, *DataEND*, or *Trigger* message then the remote/local request should be deferred until after the message (on synchronous channel) with the designated MessageID was processed.

The server shall send the *AsyncRemoteLocalResponse* after any server defined actions are complete, however it shall not wait for locks granted to other clients to be released. Although the *AsyncRemoteLocalResponse* is sent immediately if another client is holding a lock, the server shall only act on the remote/local request after the lock is released. The behavior is device dependent.

Three logical variables maintained by the server dictate its behavior:

Remote — Controls if front panel input is accepted. Note that remote input is always accepted. If Remote is true, front panel input is not accepted, with the exception of the local key. If the local key is pressed and LocalLockout is set to false, Remote is set false so that subsequent front panel input is accepted. When the connection is initialized Remote is false.

RemoteEnable — Mimics the GPIB REN line, but is maintained by the individual server. When the connection is initialized RemoteEnable is true.

LocalLockout — If true, the front panel local key has no affect. If false, the front panel local key sets Remote to false. When the connection is initialized LocalLockout is false.

[5] The local key, as defined by IEEE 488, is a key on the instrument an operator can use to gain front panel access to the instrument when front panel access has been automatically disabled by the 488 protocol.

If RemoteEnable is true and new data or control information arrives via the HiSLIP protocol, Remote is set to true. Specifically, any of the following messages on the synchronous channel set remote to true, provided that a deferred implication is processed:

- *Data*
- *DataEND*
- *Trigger*

Or any of the following messages on the asynchronous channel:

- *AsyncStatusQuery*
- *AsyncDeviceClear*
- *AyncLock*

Servers are permitted to take a device specific action for *VendorSpecific* messages.

RemoteLocal HiSLIP messages set these state variables as described in Table 25. In that table, T indicates the variable is set, F indicates the variable is cleared, and nc indicates the variable is not changed.

After the Initialization Transaction and Device Clear Transaction, the client shall use the MessageID = 0xfffffefe (0xffffff00-2) in the *AsyncRemoteLocalControl* message.

The remote/local control codes correspond to the parameters of the VISA viGpibControlREN[6] function call. The behavior is chosen to emulate the behavior of a GPIB device. [7]

[6] The VISA specification (vpp43, Table 6.5.1) specifies the following:

Mode	Action Description
VI_GPIB_REN_DEASSERT	Deassert REN line.
VI_GPIB_REN_ASSERT	Assert REN line.
VI_GPIB_REN_DEASSERT_GTL	Send the Go To Local command (GTL) to this device and deassert REN line.
VI_GPIB_REN_ASSERT_ADDRESS	Assert REN line and address this device.
VI_GPIB_REN_ASSERT_LLO	Send LLO to any devices that are addressed to listen.
VI_GPIB_REN_ASSERT_ADDRESS_LLO	Address this device and send it LLO, putting it in RWLS.
VI_GPIB_REN_ADDRESS_GTL	Send the Go To Local command (GTL) to this device.

[7] The VISA API provides general control of GPIB that is not necessary for a HiSLIP client. Practical HiSLIP applications can be handled by using just three values for the mode: VI_GPIB_REN_DEASSERT which will always place the instrument in local, VI_GPIB_REN_ASSERT_ADDRESS_LLO which will always put the instrument into remote with local-lockout, and VI_GPIB_REN_ASSERT_ADDRESS which will place the instrument into remote, but enable the front panel local key (with automatic transitions back to remote when remote data is received). Unfortunately, the names of these modes are not very mnemonic.

Table 25 Remote Local Control Transactions

Control Code (request)	Corresponding VISA mode from viGpibControlREN	Behavior		
		RemoteEnable	LocalLockout	Remote
0 – Disable remote	VI_GPIB_REN_DEASSERT	F	F	F
1 – Enable remote	VI_GPIB_REN_ASSERT	T	nc	nc
2 – Disable remote and go to local	VI_GPIB_REN_DEASSERT_GTL	F	F	F
3 – Enable remote and go to remote	VI_GPIB_REN_ASSERT_ADDRESS	T	nc	T
4 – Enable remote and lock out local	VI_GPIB_REN_ASSERT_LLO	T	T	nc
5 – Enable remote, got to remote, and set local lockout	VI_GPIB_REN_ASSERT_ADDRESS_LLO	T	T	T
6 – go to local without changing state of remote enable	VI_GPIB_REN_ADDRESS_GTL	nc	nc	F

If multiple clients make changes the behavior shall be the same as if a single client made all the requests serially in whatever order the requests are handled by the server.

On closing the connection, the remote local behavior is defined by the server.

6.8 Trigger Message

Table 26 Trigger Message

Step	Sender	Message	Action
1	Client	<*Trigger*><RMT-delivered><MessageID><0>	Initiate a trigger

The trigger message is used to emulate a GPIB Group Execute Trigger. This message shall have the same instrument semantics as GPIB Group Execute Trigger.

The fields in the *Trigger* message are:

RMT-delivered	RMT-delivered is 1 if this is the RMT-delivered flag-carrying message since the HiSLIP client delivered an RMT to the client application layer.
MessageID	MessageID identifies this message so that response data from the server can indicate the message that generated it. For generation of the MessageID, see sections 3.1.2 (*Synchronized Mode Client Requirements*) and 3.2.2 (*Overlap Mode Client Requirements*).

6.9 Vendor Defined Transactions

Table 27 Vendor Defined Transaction

Step	Sender	Message	Action
1	Either	<*VendorDefined*><arbitrary><arbitrary ><length><payload>	Vendor defined
2		Response – if unrecognized non-fatal error, if recognized vendor defined.	

VendorDefined messages may be used arbitrarily by vendors on either the synchronous or asynchronous channels. Clients or servers that do not recognize *VendorDefined* messages shall ignore the message including the number of subsequent data bytes.

Devices or Servers receiving VendorDefined commands they do not support shall respond with an *Error* message on the same channel the Vendor defined message arrived on specifying “Unrecognized Vendor Defined Message”.

6.10 Maximum Message Size Transaction

Table 28 Maximum Message Size Transaction

Step	Sender	Message	Action
1	Client	<*AsyncMaximumMessageSize*><0><0><8><8-byte size>	The server sets the maximum message size it will send to the client to the specified value
2	Server	<*AsyncMaximumMessageSizeResponse*><0><0><8><8-byte size>	The client sets the maximum message size it will send to the server to the specified value

The *AsyncMaximumMessageSize* transaction is used to inform the client and server of the maximum message size they are permitted to send to the other one on the synchronous channel. This is especially important for small devices that may be unable to handle large messages.

The *AsyncMaximumMessageSize* transaction is initiated by the client. Neither clients nor servers are obligated to accept a particular message size beyond what is necessary during initialization. Therefore it is prudent for clients to initiate this transaction as part of initialization to inform the server of its message size limitations and determine the server limitations.

The specified message sizes only apply to the synchronous channel.

The 8-byte buffer size is sent in network order as a 64-bit integer.

Servers shall keep independent client message sizes for each HiSLIP connection.

If the server has announced that encryption is mandatory or the initial connection is required to be encrypted, this transaction may be performed between the Initialization Transaction and the Establish Secure Connection Transaction.

Note that the maximum message size shall not apply to the Establish Secure Connection Transaction and split the payload of e.g. *GetSaslMechanismListResponse* or *AuthenticationExchange*.

6.11 Interrupted Transaction

Table 29 Interrupted Transaction

Step	Sender	Message	Action
1	Server	<*AsyncInterrupted*><0><MessageID><0>	Clear buffered messages.
2	Server	<*Interrupted*><0><MessageID><0>	Clear buffered messages.

The interrupted transaction is sent from the server to the client when the server detects an interrupted protocol error. The client shall clear any buffered *Data, DataEND,* or *Trigger* messages from the server and ignore any subsequent *Data, DataEND,* or *Trigger* messages until it has received both the synchronous *Interrupted* message and asynchronous *AsyncInterrupted* messages arrive.

The MessageID field indicates the MessageID of the *Data, DataEND,* or *Trigger* message that interrupted the server response.

For usage of the interrupted transaction, see section 3 (*Overlapped and Synchronized Modes*).

6.12 Device Clear Transaction

Device clear clears the communication channel.

Table 30 Device Clear Complete Transaction

Step	Sender	Message content	Action
1	Client	<*AsyncDeviceClear*><0><0><0>	
----	Client	complete messages underway and abandon any pending messages	Abandon pending messages and wait for in-process synchronous messages to complete
2	Server	<*AsyncDeviceClearAcknowledge*><featurePreference><0><0>	The client shall wait for this acknowledgement before additional processing.
3	Client	<*DeviceClearComplete*><featureRequest><0><0>	Indicate to server that synchronous channel is cleared out.
4	Server	NA	Upon receipt of the *AsyncDeviceClear* messages abandon any operations in progress.
5	Server	NA	Disregard input messages until the *DeviceClearComplete* message is found. But continue to require well-formed messages.
6	Server	<*DeviceClearAcknowledge*><featureSetting><0><0>	Client and server each resume normal operation.

To send a device clear, the client will:

1. Finish sending any partially sent messages on either channel.
2. Send the *AsyncDeviceClear* message on the asynchronous channel.
3. If the protocol was amidst any of the following transactions permit them to complete if the server responds before sending *DeviceClearAcknowledge*
    a. *Lock (client waiting for AsyncLockResponse)*

    If *DeviceClearAcknowledge* arrives from the server before these other operations are acknowledged, the client HiSLIP shall assume the operations were not completed.
4. Clear messages on the synchronous and asynchronous channels with the exception of *FatalError*, *DeviceClearAcknowledge*, and *AsyncDeviceClearAcknowledge*.
5. Wait for the *AsyncDeviceClearAcknowledge* message.
6. Send the *DeviceClearComplete* message on the synchronous channel indicating to the server that no further messages will be sent to it.
7. Wait for the server to respond with *DeviceClearAcknowledge* on the synchronous channel.
8. *The MesssageID is reset to 0xffff ff00.*
9. Resume normal operation.

When the server receives the asynchronous *AsyncDeviceClear* message, it shall:

1. Complete any partially complete transactions involving the asynchronous channel without waiting for timeouts.
2. Send *AsyncDeviceClearAcknowledge.*
3. Finish sending any partially sent messages to the client. Complete with the normal behaviors, without waiting for any timeouts.
4. Abandon any buffered unsent transactions.

5. Clear any well-formed messages received from the client on the synchronous channel.
6. Accept and ignore subsequent synchronous messages until it finds the synchronous *DeviceClearComplete* message.
7. Send *DeviceClearAcknowledge* message back to the client (after the above steps are complete and it has received the device clear complete message).
8. Resume normal operation.

If at any time during device clear management either the client or server encounter a poorly formed message they shall send a *FatalError* message and do the *FatalError* processing.

If at any time during the device clear management the client or server determines that the other device is not responding in a timely fashion, it shall send a *FatalError* message and do the *FatalError* processing. The determination of an appropriate time may vary with the application, 40 to 120 seconds are reasonable values.

6.12.1 Feature Negotiation

During device clear, the features listed in Table 31 are negotiated between the client and server. The features are specified through a feature bitmap that is sent in the control code of three different messages.

The feature negotiation occurs in three steps:

1. The server proposes values that it prefers with the *AsyncDeviceClearAcknowledge* message.

2. The client indicates values that it requests in the *DeviceClearComplete* message.

3. The server indicates the values that both client and server will use in the *DeviceClearAcknowledge* message.

The server shall identify the default values it prefers for the features in the *AsyncDeviceClearAcknowledge* message. Servers shall support any such capabilities that it requests.

The server shall accept the value proposed by the client in the *DeviceClearComplete* message if it is capable of supporting them.

The client shall use the values specified by the server in the *DeviceClearAcknowledge* message.

Note that the *InitializeResponse* has the server preferred features specified in it.

Table 31 Features negotiated during device clear

Control Code Bit	Name	Meaning
0	Overlapped	Related to current connection False- Synchronized mode True - Overlapped mode
1	Encryption mode	Related to logical instrument specified by sub-address of the Initialize Message False - Encryption mode optional or negotiated HiSLIP Version < 2.0 True - Encryption mode mandatory
2	Initial encryption	Related to logical instrument specified by sub-address of the Initialize Message False - Establish Secure Connection Transaction not required after Initialization Transaction True - Establish Secure Connection Transaction must follow Initialization Transaction.

6.13 Service Request

Table 32 Service Request

Step	Sender	Message content	Action
1	Server	<*AsyncServiceRequest*><status><0><0>	Client initiated request for service

The server requests service by sending the *AsyncServiceRequest* message.

The control code contains the 8-bit status register.

No values, including rqs (request service), are cleared in the status register. Note that since no values are cleared, the client must do an *AsyncStatusQuery* to clear the rqs bit and enable additional *AsyncServiceRequest* messages.

This message is not acknowledged.

6.14 Status Query Transaction

Table 33 Status Message

Step	Sender	Message content	Action
1	Client	<*AsyncStatusQuery*><RMT-delivered><MessageID><0>	Client initiates a request for status
2	Server	<*AsyncStatusResponse*><status><0><0>	Status information is sent back in the control code field

The status query provides an 8-bit status response from the server that corresponds to the VISA viReadSTB operation. The status query is initiated by the client and sent on the asynchronous channel.

The calculation of the message available bit (MAV) of the status response differs for overlapped and synchronized modes and requires the client to provide a different Message ID.

The following are the fields of the *AsyncStatusQuery* client message:

RMT-delivered — RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the client delivered RMT to the application layer. Note that RMT-delivered is only reported once.

MessageID — In synchronized mode, this field contains the MessageID of the most recent *Data, DataEND,*or *Trigger* message sent by the client.

In overlapped mode, this field contains the MessageID of the most recent *Data* or *DataEND* message delivered to the client application layer.

The following are the fields of the *AsyncStatusResponse* server message:

status — This field contains an 8-bit status response from the server.

When clients send *AsyncStatusQuery* messages, they shall set the message header to the MessageID field of the message header of the most recently sent *Data, DataEND,* or *Trigger* message. See section 3, *Overlapped and Synchronized Modes* for the server construction of the status response.

After the Initialization and Device Clear transactions, the client shall use the MessageID = 0xfffffefe (0xffffff00-2) in the *AsyncStatusQuery* message.

6.14.1 MAV Generation in Synchronized Mode

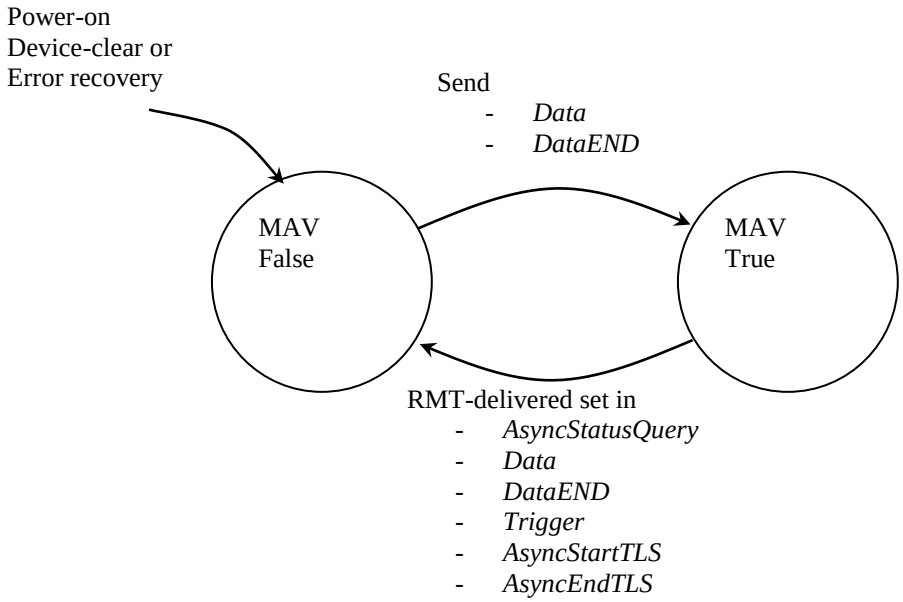


Figure 1 MAV Generation in Synchronized Mode

HiSLIP asynchronously reads the status back from the server using the *AsyncStatusQuery* message on the asynchronous channel. However, IEEE Std 488.2 requires servers to include a MAV (Message Available) bit in position 4 that indicates if data is available from the server. The RMT-delivered indication sent from the client to the server assists the server in determining the correct value for MAV.

Per IEEE Std 488.2 the MAV bit shall be sent in bit 4 (zero-based) of the status response from the server.

MAV shall be set true when the server sends the first *Data* or *DataEND* of a response.

MAV shall be set false when RMT-delivered is indicated by the client in any RMT-delivered flag-carrying message.

Figure 1 shows how the MAV shall be calculated.

Note that new-reason-for-service is only asserted on the transitions between these states. Therefore, an *AsyncServiceRequest* is only generated for MAV once per message.

6.14.2 MAV Generation in Overlapped Mode

In overlapped mode, the server shall compare the MessageID specified in the *AsyncStatusQuery* to the current MessageID counter. If any messages have not been fully delivered to the client application, MAV shall be set true.

6.14.3 Implementation Note

In some conditions, TCP may deliver an *AsyncStatusQuery* before it delivers a preceding *Data/DataEND* or *Trigger* message generated by the client.

In synchronized mode, the server should consult the MessageID provided with the *AsyncStatusQuery*. If this MessageID is not equal to the MessageID of the last received *Data*, *DataEND*, or *Trigger* message then MAV shall be set false. Note that in this special case the server either has no data for delivery or will be interrupted with the next pending synchronous message. In the case where MAV is reported as false because of a pending interrupted error, it is not necessary for the server to detect and report the error in this status response however the interrupted error will be reported subsequently with normal interrupted processing.

In overlapped mode, the provided MessageID strictly indicates the availability of new data. Therefore, the client should never presume that the absence of message available indicates additional data will not be made available later.

6.15 Establish Secure Connection Transaction

This transaction is only available if the Secure Connection capability is supported.

The purpose of this transaction is to encrypt the connection and to authenticate server and client mutually. If encryption is mandatory or the server requires the initial connection to be encrypted, this transaction must follow immediately after the Initialization Transaction or after Maximum Message Size Transaction. Otherwise, the server declares fatal error code 5. If encryption is optional, this transaction can be started at any time if a secure connection has not been established.

Table 34 Establish secure connection transaction

Step	Initiator	Message content	Action
0	Client	<*AsyncStartTLS*><RMT-delivered><MessageIDsent><MessageIDreceived>	
1	Server	<*AsyncStartTLSResponse*><0=busy,1=success,3=error><0><0>	Based on the RMT and message IDs from step 0, the server determines if it is ready to switch the encryption mode (indicated by success), if it is busy (this could happen if there is still data in the I/O buffers), or if there is an error. This transaction continues only if success is indicated.
2	Both	TLS handshaking according to Chapter 4 Handshake Protocol (RFC 8446) on the asynchronous channel	After successful handshaking the asynchronous connection is encrypted
3	Client	<*StartTLS*><0><0><0>	
4	Both	TLS handshaking according to Chapter 4 Handshake Protocol (RFC 8446) on the synchronous channel	After successful handshaking the synchronous connection is encrypted
5	Client	<*GetSaslMechanismList*><0><0><0>	The client requests the available SASL mechanisms. This step is optional, the client may choose to continue with step 7 instead.
6	Server	<*GetSaslMechanismListResponse*><0><0><mechanisms list>	The Server sends a space-separated list containing the available SASL authentication mechanisms.
7	Client	<*AuthenticationStart*><0><0><Mechanism>	The Client selects an authentication method.
8	Client	<*AuthenticationExchange*><0><0><data>	The client sends the initial response based on the chosen mechanism. If the mechanism requires a challenge before the first response, this message contains no data.
9	Server	<*AuthenticationExchange*><0><0><data>	The server sends a challenge or continues with step 11.
10	Client	<*AuthenticationExchange*><0><0><data>	The client sends a response to the challenge and waits for either another challenge or the authentication result from the server. The server continues with step 9.
11	Server	<*AuthenticationResult*><0|1><error code><data>	The server sends the result of the authentication. If authentication was not successful, the client must either return to step 6, close the connection, or continue with End Secure Connection Transaction (only allowed if encryption is optional). If successful, the secure HiSLIP connection is ready for use.

The following are the fields of the *AsyncStartTLS* client message:

- *RMT-delivered*: RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the client delivered RMT to the application layer. Note that RMT-delivered is only reported once.
- *MessageIDsent*: this field contains the MessageID of the most recent *Data*, *DataEND* or *Trigger* message sent by the client. For vendor specific messages it is up to the vendor if the message ID of the last vendor specific message should be used here.
- *MessageIDreceived*: this 4-byte payload contains the MessageID of the most recent *Data* or *DataEND* message delivered to the client application layer.

After the Initialization and Device Clear transactions, the client shall use the MessageIDsent = 0xfffffefe (0xffffff00-2) and MessageIDreceived = 0xfffffefe (0xffffff00-2) in the *AsyncStartTLS* message.

By comparing the values of MessageIDsent and MessageIDreceived in the *AsyncStartTLS* message to the value the server knows from its last outgoing message, the server determines whether its and the client's buffers are empty. Based on the result the server indicates its status:

- *Success*: All buffers are empty and it is possible to switch the encryption mode. The transaction continues normally.
- *Busy*: The buffers are not empty. This transaction is aborted. Note, that a Device Clear Transaction empties the buffers so that a following switch of the encryption mode will succeed.
- *Error*: There is an error preventing to switch the encryption mode. This transaction is aborted. The encryption mode cannot be switched even after a Device Clear Transaction. One reason could be, that encryption is already switched on. The TLS last error descriptor provides further details.

After the client receives the *AsyncStartTLSResponse* message indicating success (step 1) it shall continue with the "Client Hello" message according to TLS handshake (RFC 8446) on the asynchronous channel. After the client sends the *StartTLS* message (step 3), it shall continue with "Client Hello" on the synchronous channel.

For a successful connection, during the TLS handshake the server must provide a certificate, which has not yet expired, which has not been revoked and which is signed by a certificate authority trusted by the client. If one of these criteria is not fulfilled, the client declares fatal error code 5 and closes the connection.

A server that supports SASL authentication mechanism EXTERNAL shall request a client certificate according to RFC 8446 4.3.2. If the client chooses a different mechanism, the client may ignore the request and not provide a certificate. In the TLS handshake, in this case, only the server is required to provide a certificate. If EXTERNAL is the only mechanism supported by the server, the client shall provide a certificate or the authentication fails.

The server shall support at least one SASL authentication mechanism.

The list of supported SASL mechanisms sent by the server in the *GetSaslMechanismListResponse* message should be ordered by preference. The first mechanism in the list should be the one with the highest preference by the server, whereas the last mechanism in the list should be the one with least preference. If the client does not have a preference, it should attempt the mechanisms in the preference order of the server.

If an unsupported mechanism is declared in the *AuthenticationStart* message, the server declares the non-fatal error code 5.

Client authentication is performed with dedicated HiSLIP messages (steps 7 to 11).

If the client or server transmit invalid data in the *AuthenticationExchange* messages, the other side shall declare the fatal error code 5.

It is recommended to reuse the session key, which was negotiated in the TLS handshake of the asynchronous channel, in the TLS handshake of the synchronous channel. This saves time when establishing the connection. Furthermore, it is recommended to resume closed sessions, according to F.1.4 of RFC 8446, if the TLS sessions or the HiSLIP session was closed.

If the authentication was not successful (step 11) the client may retry with step 7. In order to prevent brute force attacks, it is recommended that the server adds a delay for authentication after a certain number of authentication failures.

6.16 End Secure Connection Transaction

This transaction is only available if the Secure Connection capability is supported.

The purpose of this transaction is to end a secure connection. This transaction is only allowed if the Establish Secure Connection Transaction was successful and the secure connection has not been ended. If encryption mode is mandatory this transaction shall fail.

Table 35 End secure connection Transaction

0	Client	<*AsyncEndTLS*><RMT-delivered><MessageIDsent><MessageIDreceived>	
1	Server	<*AsyncEndTLSResponse*><0=busy,1=success,3=error><0><0>	Based on the RMT and message IDs from step 0, the server determines if it is ready to switch the encryption mode (indicated by success), if it is busy (this could happen if there is still data in the I/O buffers), or if there is an error. This transaction continues only if success is indicated.
2	Both	End TLS according to Chapter 6.1 RFC 8446 on the asynchronous channel	After *close_notify* is exchanged, subsequent messages on the asynchronous channel are no longer encrypted.
3	Client	<*EndTLS*><0><0><0>	
4	Both	End TLS according to Chapter 6.1 RFC 8446 on the synchronous channel	After *close_notify* is exchanged, subsequent messages on the synchronous channel are no longer encrypted.

After this transaction, the authenticity of the client and the server cannot be guaranteed.

The following are the fields of the *AsyncEndTLS* client message:

- *RMT-delivered*: RMT-delivered is 1 if this is the first RMT-delivered flag-carrying message since the client delivered RMT to the application layer. Note that RMT-delivered is only reported once.
- *MessageIDsent*: this field contains the MessageID of the most recent *Data*, *DataEND* or *Trigger* message sent by the client. For vendor specific messages it is up to the vendor if the message ID of last vendor specific message should be used here.
- *MessageIDreceived*: this 4-byte payload contains the MessageID of the most recent *Data* or *DataEND* message delivered to the client application layer.

After the Initialization and Device Clear transactions, the client shall use the MessageIDsent = 0xfffffefe (0xffffff00-2) and MessageIDreceived = 0xfffffefe (0xffffff00-2) in the *AsyncEndTLS* message.

By comparing the values of MessageIDsent and MessageIDreceived in the *AsyncEndTLS* message to the value the server knows from its last outgoing message, the server determines whether its and the client's buffers are empty. Based on the result the server indicates its status:

- *Success*: All buffers are empty and it is possible to switch the encryption mode. The transaction continues normally.
- *Busy*: The buffers are not empty. This transaction is aborted. Note, that a Device Clear Transaction empties the buffers so that a following switch of the encryption mode will succeed.

- *Error*: There is an error preventing to switch the encryption mode. This transaction is aborted. The encryption mode cannot be switched even after a Device Clear Transaction. One reason could be, that encryption is already switched on. The TLS last error descriptor provides further details.

After the client receives the *AsyncEndTLSResponse* message indicating success (step 1) it shall continue with the "close_notify" message according to TLS handshake (RFC 8446) on the asynchronous channel. After the client sends the *EndTLS* message (step 3), it shall continue with "close_notify" on the synchronous channel.

After ending the secure connection, both the server and the client should store the session key, for being able to resume the TLS session.

A. Analysis of Interrupted Conditions

This is an informative appendix and is not part of the HiSLIP standard requirements.

The following transaction diagrams describe HiSLIP behavior in synchronized mode with various interrupted conditions.

A.1 Slow Client

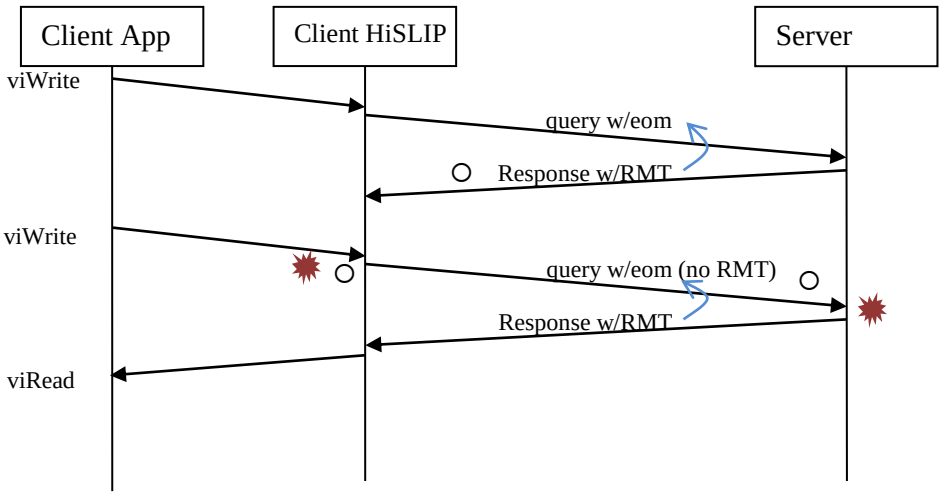


Figure 2 Interrupted error with slow client

Detailed explanation:

1. At this point the response indicates the eom from the preceding message.
2. Note that the client probably will not get a chance to execute until the client application calls viWrite. However, the client HiSLIP has the opportunity to take care of input processing before attempting to send the second write message. At this point, the client detects the error based on section 3.1.2 rule 3 and clears the first response from its buffer. (this error detection is essential at this point to ensure that the buffered response is not provided to a subsequent viRead). The client then sends the second query normally.
3. Note that the second query indicates that the RMT was NOT delivered to the application layer. Therefore the server will also detect the error based on 3.1.1 rule 2. Since the last action by the server was to send RMT, no error handling is necessary other than reporting the error.

A.2 Fast Client

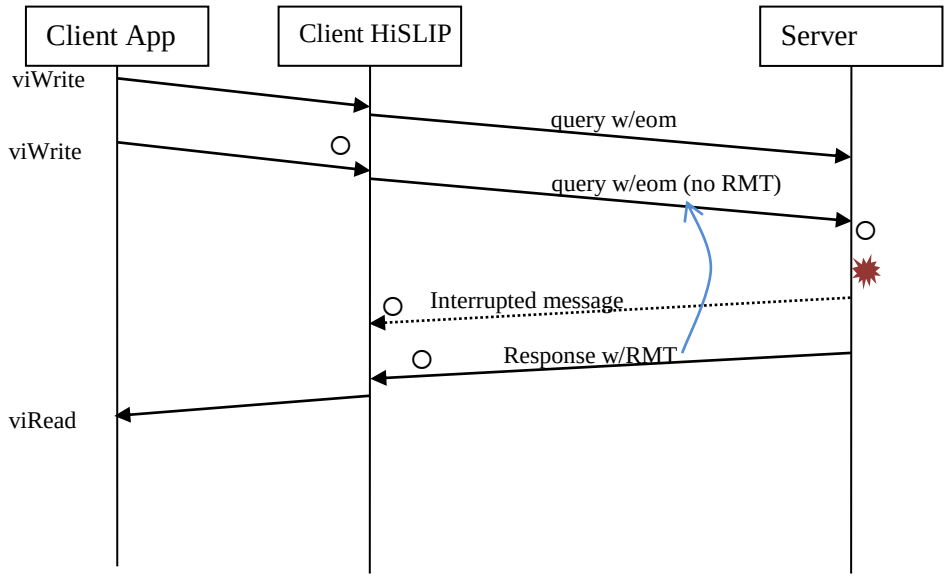


Figure 3 Interrupted error with fast client

Detailed explanation:

1. Note that from the perspective of the client HiSLIP, there may not be a problem, since the data being written may be commands.
2. Per section 3.1.1 rule 1 The server detects interrupted because it has a complete response (with RMT) and the input buffer is not empty. The response to the first query is never sent.
3. The response to the second query is sent normally.
4. NOTE – although not required here, the server sends an interrupted message (as shown in *Figure 4 Interrupted error with fast client and partial response*).

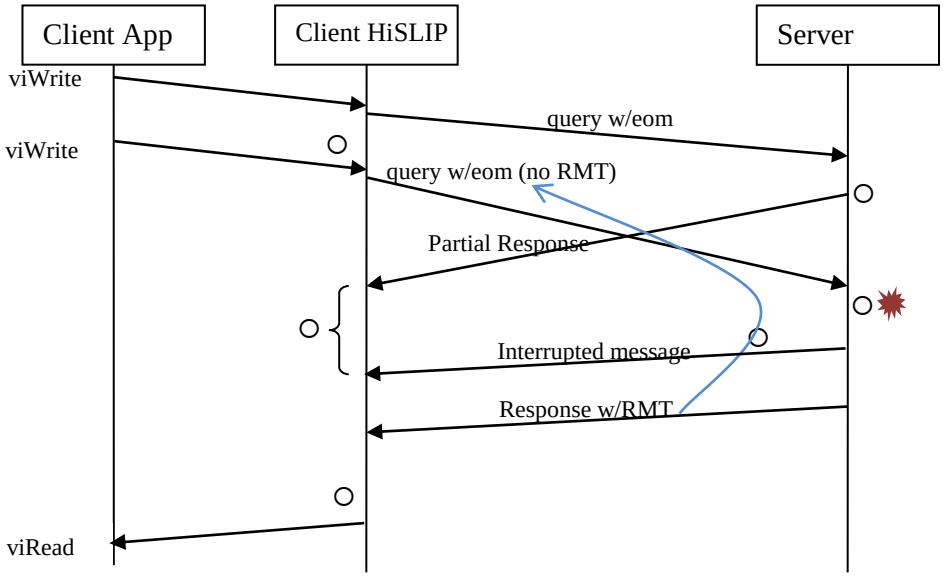


Figure 4 Interrupted error with fast client and partial response

Detailed explanation:

1. Note that from the perspective of the client HiSLIP, there may not be a problem, since the data being written may be commands.
2. Per IEEE Std 488.2, the server chooses to send a partial response (without RMT) to the client. IEEE Std 488.2 permits delivering this to the client, but the RMT corresponding to the first query must not be delivered.
3. Per section 3.1.1 rule 1, the server detects interrupted because it has a complete response (with RMT) and the input buffer is not empty (note the server is still completing processing on the first query). The final portions of the response to the first query are not sent.
4. Server informs the client that the partial response should be cleared if not already delivered.
5. The partial response is only delivered to the client if the client application attempts to read before the *Interrupted* or *AsyncInterrupted* message arrives and is detected by the client protocol. In this illustration, the HiSLIP client will not deliver the partial response.
6. The response to the second query is sent normally.

A.3 Intermediate Timing

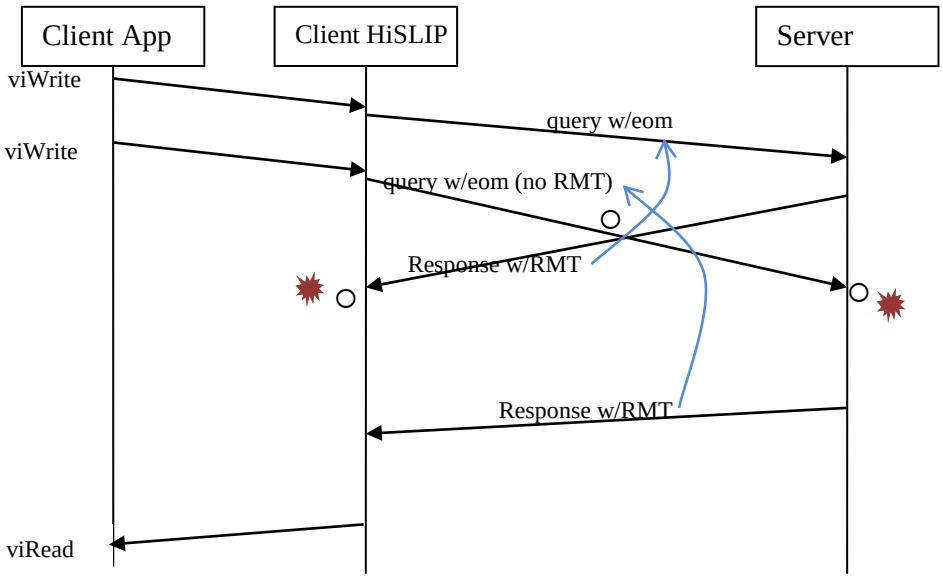


Figure 5 Race condition: first response and second query pass in-flight

Detailed explanation:

1. Second query and first response cross in-flight. Note that it is essential to have some error detection on the client in this case, otherwise this errant response would be delivered.
2. Based on section 3.1.1 rule 2, the server detects the error and reports it. Since it has already launched the first response it takes no additional action.
3. Based on section 3.1.2 rule 1, the client detects the stale response and clears it without offering any to the client app.