Interchangeable Virtual Instruments

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

October 29, 2009
Revision 0.31

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

Trademarks

Table of Contents

IVI HiSLIP Revision History

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

Table 1-1. IVI HiSLIP Class Specification Revisions

Status	Action
Revision 0.30	Brings this document in-line with conclusions from 2009-October Longmont meeting and preceding meetings: 1. Initialization sequence didn't handle spec major/minor correctly. 2. Align the remote/local with decision. That is, remove the async acknowledge operation. 3. Clarify the use of an ASCII string in a payload (no null in count or transmitted) 4. Added overlap mode. a. Added to *InitializeResponse* b. Added to device clear protocol (added 3.11.1) c. Status calculation of MAV : 3.13.3 d. General discussion of interrupted and protocol requirements throughout section 4 5. Lock changes a. Moved to async b. Added MessageID to unlock c. Changed counting to binary d. Added section "unlock considerations" to 3.5 6. Fixed broken links in appendix
Revision 0.31	Per team meeting 10/29 1. Added two lock release types: shared and exclusive 2. Specified that clients must implement both overlapped and synchronized and servers may implement either or both. 3. Added comment to the initialize sequence explicitly stating that it is legitimate fro the client to abandon the initialize after receiving *InitializeSync* instead of establishing the async connection. 4. Added statement in 2.1 that locks are dropped when the connection closes

1 Overview of the IVI HiSLIP Specification

HiSLIP is a protocol for TCP-based instrument control that provides the capabilities of conventional test and measurement protocols with minimal impact to performance. A HiSLIP instrument performs with approximately the performance of a simplistic TCP implementation. The HiSLIP protocol includes:

- Device clear
- Instrument status reporting with message available calculation per IEEE 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

1.1 IVI HiSLIP Overview

HiSLIP creates two TCP connections to the same server port 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).

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).

1.2 References

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

VXIPNP	VXIplug&play VISA Specification defines the API between the client application and the client-side HiSLIP implementation.
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 488.2 defines the interrupted protocol requirements as well as the appropriate server behavior for several of the GPIB messages.

1.3 Definitions of Terms and Acronyms

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

RMT	From IEEE 488.2: Response Message Terminator. The NL^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 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. It is a special capability that must be provided by the protocol.
eom	From IEEE 488.2: end-of-message. The termination character of a message to the server. The eom is always new-line, END, or a new-line followed by an END. For the purposes of this analysis, it is implicit after group execute trigger.

interrupted	From IEEE 488.2: A protocol 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.

2 HiSLIP Message Format

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.

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. With 'H' in the most significant network order position and 'S' in the second byte.
Message Type	This field identifies this message. See Table 3 for a description of the HiSLIP messages. See Table 26 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, 0 shall be used.
MessageParameter	This 32-bit field has various uses in different messages.
Payload Length	This field indicates the length in octets of the payload data contained all messages. This field is a 64-bit integer. The maximum data transfer size may be limited by the implementation, see 3.9, *Maximum Message Size Transaction* for details. If the message type does not use a payload, the length shall be set to zero.

All Hi-SLIP fields are marshaled onto the network in network order (big endian). That is, most significant byte first.

Where the specification call for an ASCII string as the payload the payload length shall refer to the length of the number of characters in the string. A trailing NUL character shall not be sent.

Table 3 HiSLIP Messages

Sender	Channel	Message Type	Control Code (1 byte)	Message Parameter (4 bytes)	Payload
C	S	*Initialize*	--	UpperWord : Client protocol version LowerWord : Client-vendorID	sub-address (ASCII)
S	S	*InitializeResponse*	Bit 0 : 1 Prefer Overlap : 0 Prefer Synchronized	UpperWord : Server Protocol version LowerWord : SessionID	-
E	E	*FatalError*	ErrorCode (see Table 7)	--	Error Message in ASCII
E	E	*Error*	ErrorCode (see Table 10)	--	Error Message in ASCII
C	S	*Data*	0 – RMT was not delivered 1 – RMT was delivered	MessageID (this msg)	data
S	S		-	MessageID of message generating this response or zero	data
C	S	*DataEND*	0 – RMT was not delivered 1 – RMT was delivered	MessageID (this msg)	data
S	S			MessageID of message generating this response (required)	data
C	S	*AsyncLock*	1 – Request	Timeout (in ms)	LockString in ASCII, may be of zero length.
			0 – Release	MessageID	--
S	A	*AsyncLockResponse*	0 – lock refused 1 – lock granted	--	--
C	S	*RemoteLocalControl*	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	--	--
C	A	*AsyncDeviceClear*	--	--	--

S	A	*AsyncDeviceClearAcknowledge*	Feature-bitmap	--	--
C	S	*DeviceClearComplete*	Feature-bitmap	--	--
S	S	*DeviceClearAcknowledge*	Feature-bitmap		
C	S	*Trigger*	0 – RMT was not delivered 1 – RMT was delivered	MessageID (this message)	--
S	S	*Interrupted*	--	MessageID	--
S	A	*AsyncInterrupted*	--	MessageID	
C	S	*MaximumMessageSize*	--	--	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	A	*AsyncInitialize*	--	SessionID	--
S	A	*AsyncInitializeResponse*	--	Server-vendorID	--
S	A	*AsyncServiceRequest*	--	--	--
C	A	*AsyncStatusQuery*	0 – RMT was not delivered 1 – RMT was delivered	MessageID of last sent message	--
S	A	*AsyncStatusResponse*	Server status response	--	--
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 send on either the synchronous or asynchronous channel

2.1 Locking

After the initialization transaction, all subsequent HiSLIP messages from the client to the server are subject to access control governed by the locking transaction.

Section 3.5 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. Any messages other than Lock sent on either the synchronous or asynchronous connection will be acknowledged with the Error message with the error “Operation ignored because a different client has locked the server”.

If a client is holding an exclusive lock, only that client is permitted access to the server. If no client is holding an exclusive lock, then any client holding a non-exclusive lock 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 non-exclusive lock, only a client holding a non-exclusive lock shall be granted an exclusive lock. However, if no client is holding a non-exclusive 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 session should be taken to require that a server is not permitted to implement other security mechanisms that may result in it refusing to grant access to any client.

3 HiSLIP Transactions

The following sections describe the HiSLIP protocol transactions.

3.1 Initialization Transaction

The purpose of the initialization procedure is to establish the HiSLIP connection to 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.

Server's shall support multiple simultaneous clients initializing.

Table 4 Initialization Transaction

Step	**Initiator**	**Message content**	**Action**
0	Server	<none>	Server passively opens TCP connection 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><protocol version : client-vendorID><sub-address>	Client starts the initailization by identifying the vendor, specifying the subaddress, and advertising the protocol version it supports.
3	Server	<*InitializeResponse*><0><server version : SessionID><0>	Server response with its protocol version. The two go to the lower of server version and client version. The server also provides the SessionID to send with Initialize Async.
5	Client	Opens the asynchronous TCP connection (TCP SYN message)	Client opens second connection for the asynchronous channel on same server port
6	Client	<*AsyncInitialize*><*0*><SessionID><0>	The client sends SessionID to associated this TCP session with the established HiSLIP synchronous channel.
7	Server	<*AsyncInitializeResponse*><0><server-vendorID><0>	Server acknowledges initialize and provides the vendor ID. The HiSLIP connection is ready for use.

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

client-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[1]. The client vendorID is send in the least significant 16-bits (big endian sense) of the 32-bit message parameter.
sub-address	This field corresponds to the VISA LAN device name. It identifies a particular device managed by this server. It is in the payload field and therefore includes a 64-bit count. The count if followed by the appropriate length ASCII sub-address. For instance: "INST0". If the sub-address is null (zero length) the initialize opens the default (perhaps only) device at this IP address.

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

server-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 send in the least significant 16-bits (big endian sense) of the 32-bit message parameter.

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-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.[2]

After the initialization sequence, the client and server will both use the highest protocol revision supported by both devices (that is, the smallest 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 initialize transaction with an appropriate *Lock* transaction. If the *Lock* operation fails then the client can close the connection.

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.

[1] 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.
[2] Ibid

3.2 Fatal Error Detection and Synchronization Recovery

Table 5 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 3.1

Table 6 Asynchronous Fatal Error Message

Initiator	**Message**	**Data Consumer**
Either client or server	Async: <*AsyncFatalError*><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 3.1

At any point, the client or server may receive a malformed message. 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 *AsyncFatalError* on the asynchronous channel with appropriate diagnostic information.

The IVI Foundation defines the error codes listed in Table 7. 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* and *AsyncFatalError* message 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 3.1).

If the error is detected by the server, after sending the *FatalError* and *AsyncFatalError*, 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 7 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
128-255	Device defined errors

3.3 Error Notification Transaction

Table 8 Synchronous Error Notification Transaction

Initiator	Message	Data Consumer
Either client or server	<*Error*><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 3.1

Table 9 Asynchronous Error Notification Transaction

Initiator	Message	Data Consumer
Either client or server	async: <*AsyncError*><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 3.1

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 10, 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 10 HiSLIP Defined Error Codes (non-fatal)

Error Code	Message
0	Unidentified error
1	Unrecognized Message Type
2	Unrecognized control code
3	Operation ignored because a different client has locked the server
4	Unrecognized Vendor Defined Message
128-255	Device defined errors

3.4 DataTransfer Messages

Table 11 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 *Data*, *DataEND* or *Trigger* 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 *Data*, *DataEND* or *Trigger* message since the client delivered RMT to the application layer. The client increments the MessageID with each *Data*, *DataEND* or *Trigger* message sent.

Table 12 Data Transfer Messages from Server to Client

Initiator	**Message**	**Data Consumer**
server	<*Data*><0><MessageID><length><data>	Accept data and handle appropriately The MessageID is the ID of the message containing the RMT that generated this response or zero.
server	<*DataEND*><0><MessageID><length><data>	Accept data and handle appropriately. Final data byte has an accompanying END. The MessageID is the ID of the message containing the RMT that generated this response.

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 *Data*, *DataEND*, *Trigger*, or *StatusQuery* 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 **Error! Reference source not found.**4.1.2 (*Synchronized Mode Client Requirements*) and 4.2.2 (*Overlap Mode Client Requirements*).

For server originated messages:

MessageID	MessageID identifies the client message responsible for generating this response. See section 4.1.1(*Synchronized Mode Server Requirements)*and 4.2.1(*Overlap Mode Server Requirements*).

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

These messages are not acknowledged.

3.5 Lock Transaction

Table 13 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.
2	Server	<*AsyncLockResponse*><0=failure, 1=success><0><0>	Response indicates if the lock was successful.

Table 14 Lock Transaction – Releasing a Lock

Step	**Sender**	**Message**	**Action**
1	Client	<*AsyncLock*><0 or 2 =release><MessageID><0>	Request (or release) lock
2	Server	<*AsyncLockResponse*><0=failure, 1=success><><0>	Response indicates if the lock was previously assigned.

The *AsyncLock* client message is used to request or release a lock as described in Table 15.

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 an appropriate indication. A timeout of 0 indicates that the server should only grant the lock if it is available immediately.

The LockString is an ASCII string.

The server shall always reply with an *AsyncLockResponse* per Table 15.

Table 15 Lock request/release operation descriptions

Lock **Control Code**	***LockResponse* Control Code**	**Description**
0 (release exclusive)	0 (fail)	Invalid attempt to release a lock that was not acquired.
0 (release exclusive)	1 (success)	Release was requested and granted.
2 (release shared)	0 (fail)	Invalid attempt to release a lock that was not acquired.
2 (release shared)	1 (success)	Release was requested and granted.
1 (request)	0 (fail)	Lock was requested but not granted
1 (request)	1 (success)	The lock was requested and granted

If a client uses a null (zero-length) LockString and it successfully attains the lock, that client is granted an exclusive lock which provides exclusive access to the server.

If the client provides a LockString and successfully attains the lock, any other client presenting the same LockString will also be granted access so long as no client is holding an exclusive lock. If multiple clients are sharing the lock it is their responsibility to provide whatever cooperating protocol is necessary at the application level.

If a client is holding an exclusive lock, only that client is permitted access to the server. If no client is holding an exclusive lock, then any client holding a non-exclusive lock is permitted access to the server. If the server has no outstanding exclusive or shared locks, then any client is permitted access to the server.

If any client is holding a non-exclusive lock, only clients holding a non-exclusive lock shall be granted exclusive locks. If no client is holding a lock, then clients requesting either exclusive or non-exclusive locks may[3] be granted those requests.

If a client already holding a shared or exclusive lock and requests it again and specifies the same LockString (including the null string) it originally used, the server will return success. This client will need to release the lock only once to give up its lock on the server. Note in this discussion client refers to a specific HiSLIP connection.

If a client already holding a lock requests it again but using a different LockString (including the null string), the redundant lock will not be granted.

3.5.1 Unlock Considerations

During an unlock operation, the control code specifies if an exclusive or shared lock should be released. Only shared locks (that is, those attained with a non-null LockString) shall be released with the release shared control code. Only exclusive locks (that is, those attained with a null LockString) shall be released with a release exclusive control code.

If one or more clients have a shared lock, and a client with an exclusive lock releases its shared lock, it will retain exclusive access to the server until it releases its exclusive lock, at which point it will no longer have either lock.

During 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

[3] Note: this permission permits servers to refuse clients based on some other security mechanism.

transmitted before the *AsyncLock* operation. The *AsyncLock* transaction will not complete the unlock until that message is complete.

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.

3.6 Remote Local Transaction

Table 16 RemoteLocal Control Transaction

Step	Sender	Message	Action
1	Client	<*RemoteLocalControl*><request><0><0>	Request remote local operation

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 when the controller needs exclusive access

The values of the request field are shown in Table 17, *Remote Local Control Transactions.*

Since it is sent on the synchronous channel, *RemoteLocalControl* will be executed after preceding *Data*, *DataEND*¸and *Trigger* messages. The server is permitted to act on the *RemoteLocalControl* immediately, or wait until after the preceding operations have been acted on by the server.

Three logical variables maintained by the server dictate its behavior:

RemoteEnable	Mimics the GPIB REN line, but is maintained by the individual server.
LocalLockout	If true, the front panel local key has no affect. If false, the front panel local key sets Remote to false.
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.

If RemoteEnable is true and new data or control information arrives via the Hi-SLIP protocol, Remote is set to true. Specifically, any of the following messages on the synchronous channel set remote true:

- *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 17. In that table, T indicates the variable is set, F indicates the variable is cleared, and nc indicates the variable is not changed.

The remote/local control codes correspond to the parameters to the VISA viGpibControlREN[4] function call. The behavior is chosen to emulate the behavior of a GPIB device. [5]

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

Mode	**Action Description**

Table 17 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.

`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.

[5] The VISA API provides general control of GPIB that is not necessary for a Hi-SLIP client. Practical Hi-SLIP 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.

3.7 Trigger Message

Table 18 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 first *Data*, *DataEND*, *Trigger*, or *StatusQuery* 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 section **Error! Reference source not found.** *(***Error! Reference source not found.**).

3.8 Vendor Defined Transactions

3.8 Vendor Defined Transactions

Table 19 Vendor Defined Transaction

Step	**Sender**	**Message**	**Action**
1	Either	<*VendorDefined*><arbitrary><arbitrary ><length><payload>	Vendor defined
		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”.

3.9 Maximum Message Size Transaction

Table 20 Maximum Message Size Transaction

Step	Sender	Message	Action
1	Client	<*MaximumMessageSize*><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 *MaximumMessageSize* transaction is used to optimize the message sizes sent between the client and server on the synchronous channel. This is especially important for small devices that may be unable to handle large messages.

The *MaximumMessageSize* transaction is initiated by the client. Neither clients nor servers are obligated to accept a particular message size beyond what is necessary during initiatlization. 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.

3.10 Interrupted Transaction

Table 21 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.

3.11 Device Clear Transaction

Device clear clears the communication channel.

Table 22 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 sync or async clear 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>	Resume normal activity

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)*
    b. *RemoteLocal (client waiting for AsyncRemoteLocalAcknowledge)*

    If *DeviceClearAcknowledge* arrives from the server before these other operations are acknowledged, the client HiSLIP shall assume the operations were not completed.
4. Accept and quietly ignore the following messages on the synchronous or asynchronous channels, including:
    a. *Error*
    b. *Data*
    c. *DataEND*
    d. *Interrupted*
    e. *AsyncInterrupted*
    f. *VendorSpecific*
    g. *AsyncMaximumMessageSizeResponse*
    h. *AsyncServiceRequest*
    i. *AsyncStatusResponse*
    j. *VendorSpecific*

    Note that *FatalError* takes precendence over device clear. If encountered in either stream the usual handling for *FatalError* shall ensue.
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.

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

1. Finish sending any partially sent messages to the client. Completing without waiting for any timeouts any pending *Lock* or *RemoteLocal* transactions.
2. Send *AsyncDeviceClearAcknowledge*
3. Abandon any buffered unsent transactions.
4. Clear any well-formed, buffered messages received from the client on the synchronous or asynchronous channels.
5. Accept and ignore subsequent synchronous and asynchronous messages until it finds the synchronous DeviceClearComplete message.
6. Send DeviceClearAcknowledge message back to the client (after the above steps are complete and it has received the device clear complete message).
7. Resume normal operation

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

3.11.1 Feature Negotiation

During device clear, the features listed in Table 23 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* messge.

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 *DeviceClearComple* message if it is capable of supporting them.

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

Table 23 Features negotiated during device clear

Control Code Bit Position	Name	Meaning

0	`Overlapped`	`False- Synchronized mode`
		`True - Overlapped mode`

3.12 Service Request

Table 24 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 status register up to 16-bits.

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

This message is not acknowledged.

3.13 Status Query Transaction

Table 25 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 a 16-bit status response from the instrument 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 *Data*, *DataEND,*Trigger or *AsyncStatusRequest* 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 a 16-bit status response from the server. Devices that only provide an 8-bit status shall use the least significant 8 bits.

When clients send *StatusQuery* 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 3.13, *Status Query Transaction,* for the server construction of the status response.

3.13.1 MAV Generation in Synchronized Mode

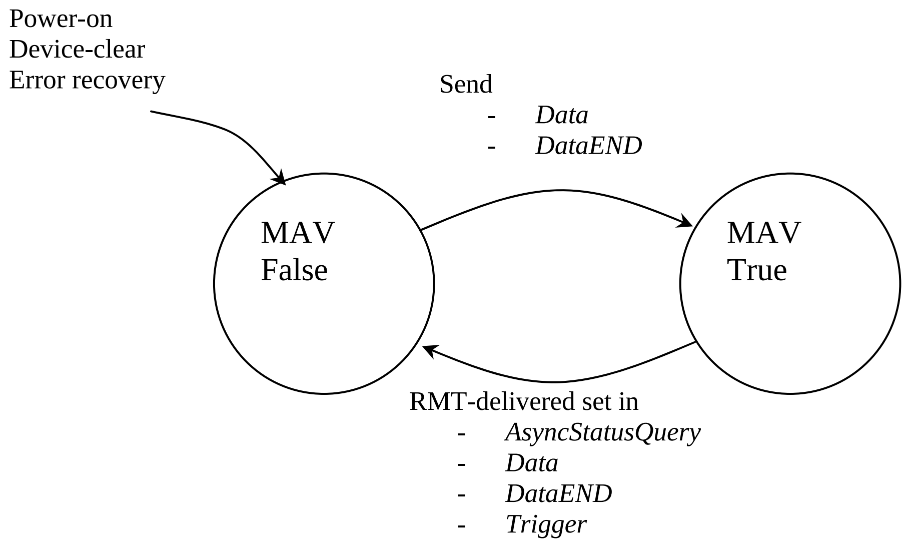

Figure 1 MAV Generation in Synchronized Mode

HiSLIP asynchronously reads the status back from the instrument using the *StatusQuery* message on the asynchronous channel. However, IEEE 488.2 requires servers to include a MAV (Message Available) bit in position 5 that indicates if data is available from the server.

Per IEEE 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 either the *Data*, *DataEND*, *Trigger* or *AsyncStatusQuery* 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, a *AsyncServiceRequest* is only generated for MAV once per message.

3.13.2 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.

3.13.3 Implementation Note

In some conditions, TCP may deliver a *AsyncStatusQuery* before it delivers a preceding *Data/DataEND* or *Trigger* message generated by the client. For many instruments this condition will not be of concern.

In synchronized mode, the server may consult the MessageID provided with the *AsyncStatusQuery* and then wait until the synchronous channel catches up to generate the status response. However, an instrument performing this delay may need to scan the input messages as they are received by TCP to identify the arrival of a message with the appropriate MessageID. This must occur even if the parser is blocked (for instance on a *WAI).

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

4 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 and the response from each will be buffered and returned to the client.
Synchronized mode	In Synchronized mode, the client is required to read the result of each query message before sending another query[6]. Failing to do so, generated the interrupted protocol error and results in the response from the preceding query being 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* transation differ between the overlapped and synchronized modes also. See section 3.13, *Status Query Transaction* for details.

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

4.1 Synchronized Mode

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

4.1.1 Synchronized Mode Server Requirements

HiSLIP servers shall implement the following:

1. When the server application layer (nominally an instrument parser) requests that HiSLIP 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 instrument error reporting mechanism to report the interrupted error within the instrument.

   - 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 *Data, DataEND, Trigger,* or *StatusQuery* 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* with RMT-deliverred flag set to true.

   When *Data, DataEND,*or *Trigger* are received, if RMT-expected and RMT-deliverred 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-deliverred are different, the instrument shall declare an interrupted error. The server shall use the instrument 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 corresponding message. In some circumstance (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.

4.1.2 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 *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 *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 initially detects *Interrupted*, 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 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.

4.2 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.

4.2.1 Overlap Mode Server Requirements

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

1. The MessageID shall be reset to 0xFFFFFF00 after device clear or initialization.

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

4.2.2 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.

5 Message Type Codes

The following table defines the numeric values of the Message Type codes.

Table 26 Message Type Value Definitions

Designation	Channel	Numeric Value (decimal)
Initialize	Synchronous	0
InitializeResponse	Synchronous	1
FatalError	Synchronous	2
Error	Synchronous	3
AsyncLock	Synchronous	4
AsyncLockResponse	Asynchronous	5
Data	Synchronous	6
DataEnd	Synchronous	7
DeviceClearComplete	Synchronous	8
DeviceClearAcknowledge	Asynchronous	9
RemoteLocalControl	Synchronous	10
Trigger	Synchronous	12
Interrupted	Synchronous	13
AsyncInterrupted	Asynchronous	14
MaximumMessageSize	Synchronous	15
AsyncMaximumMessageSizeResponse	Asynchronous	16
AsyncInitialize	Asynchronous	17
AsyncInitializeResponse	Asynchronous	18
AsyncDeviceClear	Asynchronous	19
AsyncServiceRequest	Asynchronous	20
AsyncStatusQuery	Asynchronous	21
AsyncStatusResponse	Asynchronous	22
AsyncDeviceClearAcknowledge	Asynchronous	23
VendorSpecific	Either	128-255 inclusive

6 VISA Resource Descriptor

This discussion needs to be ultimately move to the VISA specifications. It is currently in this document to facilitate increasing the visibility of the discussion and design.

The existing TCPIP resource descriptors are:

TCPIP	TCPIP[*board*][::*LAN device name*]::SERVANT
TCPIP	TCPIP[*board*]::*host address*[::*LAN device name*][::INSTR]
TCPIP	TCPIP[*board*]::*host address*::*port*::SOCKET

Note that the final token ("INSTR") is the default and implies conventional GPIB behaviors (which HiSLIP provides). Therefore, a variation on this is desirable for HiSLIP.

Note that the existing VISA resource descriptor for VXI-11 (the middle one above) requires that the LAN device name be of the form: "INST<inst_no>".

The proposal for HiSLIP is:

TCPIP[board]:: host_address[::port][::INST<inst_no>] ::HS [::INSTR]

Where:

board	identifies the NIC (network interface card) in the client computer
host_address	is an IPv4 or IPv6 address or a hostname
port	is the HiSLIP port
inst_no	this identifies a logical instrument if more than one resides at this IP address

A. Analysis of Interrupted Conditions

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

A.1 Slow Client

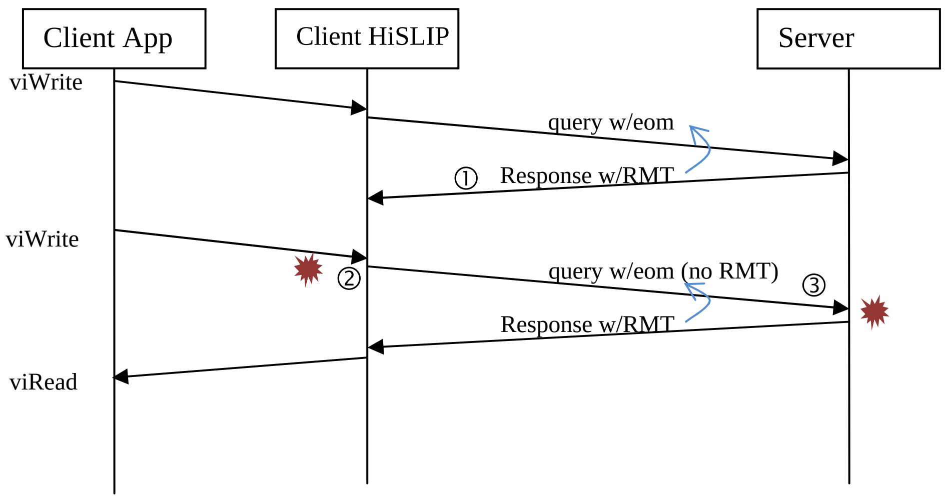

Figure 2 Interrupted error with slow client

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 4.1.2 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 4.1.1 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

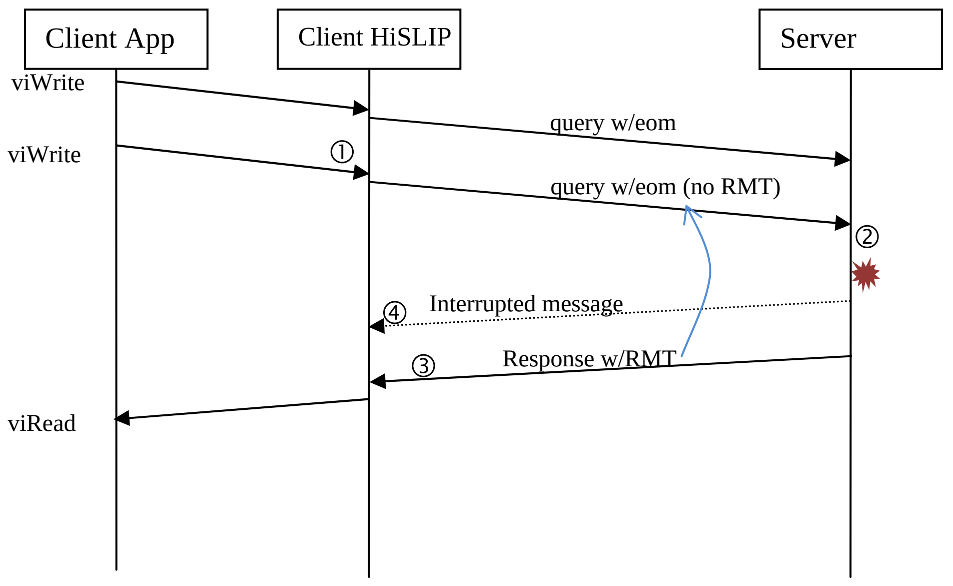

Figure 3 Interrupted error with fast client

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 4.1.1 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*).

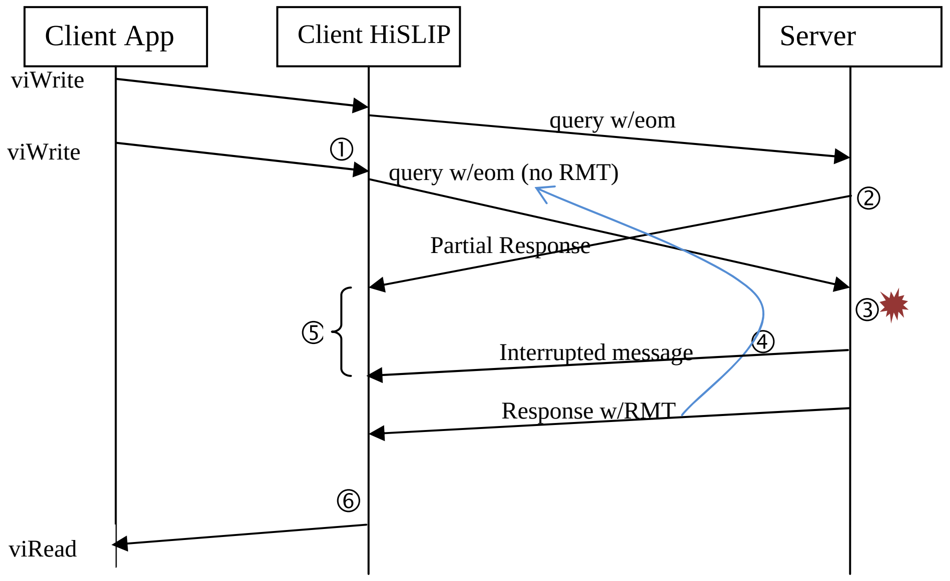

Figure 4 Interrupted error with fast client and partial response

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 488.2, the instrument chooses to send a partial response (without RMT) to the client. 488.2 permits (requires?) delivering this to the client, but the RMT corresponding to the first query must not be deliverred.
3. Per section 4.1.1 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 attempts to read before the *Interrupted* message arrives. In this illustration, HiSLIP client will not deliver the partial response.
6. The response to the second query is sent normally.

A.3 Intermediate Timing

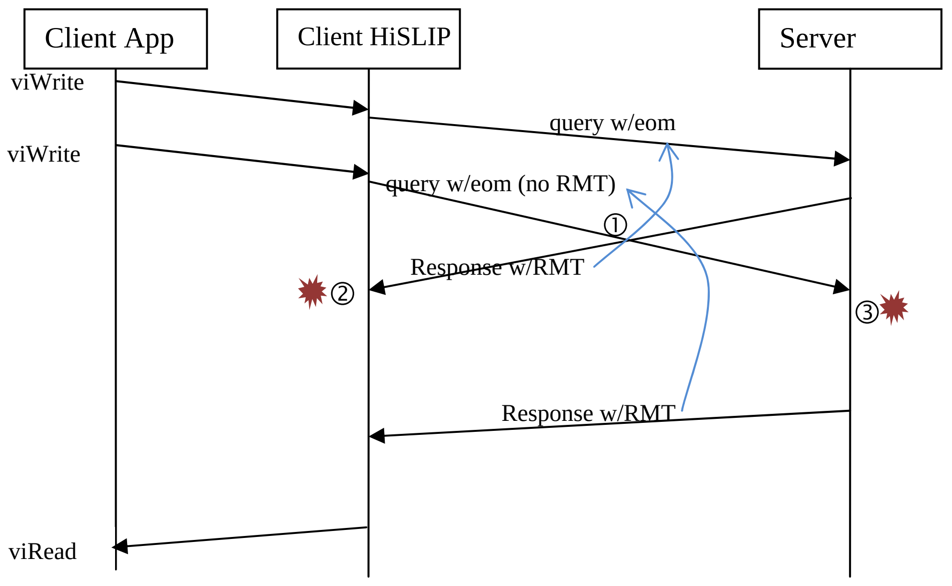

Figure 5 Race condition: first response and second query pass in-flight

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 4.1.2 rule 1, the client detects the stale response and clears it without offering any to the client app.
3. Based on section 4.1.1 rule 2, the server detects the error and reports it. Since it has already launched the first response it takes no additional action.