Working Draft

T11.1 / Project 1245-D / Rev 1.4

December 23, 1997

Information Technology - Scheduled Transfer Protocol (ST)

Secretariat : National Committee for Information Technology Standardization (NCITS)

This is an internal working draft of T11.1, a Task Group of Technical Committee T11 of Accredited Standards Committee NCITS. As such, this is not a completed standard. The contents are actively being modified by T11.1.

ABSTRACT

This document describes a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

Contacts :

T11.1 Chairman and Technical Editor	T11.1 Vice Chairman
Don Tolmie	Roger Ronald
Los Alamos National Laboratory	Raytheon E-Systems
CIC-5, MS-B255	MS 35300 HD
Los Alamos, NM 87545	PO Box 660023
Voice : 505-667-5502	Dallas, TX 75266-0023
FAX : 505-665-7793	Voice : 972-205-8043
E-mail : det@lanl.gov	FAX : 972-272-8144
	E-mail : rronald@esy.com

Reference number
ISO/IEC 11518-xx : 199x
NCITS.*** - 199x
Printed 12/23/97

Other Points of Contact:

	T11 Chairman	T11 Vice-Chairman	NCITS Secretariat
	Roger Cummings	Edward L. Grivna	NCITS Secretariat, ITI
	Distributed Processing Technology	Cypress Semiconductor	1250 Eye Street, NW Suite 200
	140 Candace Drive	2401 East 86th Street	Washington, DC 20005
	Maitland, FL 32751	Bloomington, MN 55425	
Voice :	407-830-5522 x348	612-851-5046	202-737-8888
FAX :	407-260-5366	612-851-5087	202-638-4922
E-mail :	cummings_roger@dpt.com	elg@cypress.com	ncitssec@itic.nw.dc.us

T11.1 E-mail Reflector (for HIPPI and ST technical discussions and notification of things on the web site)

Internet address for subscribing to the reflector:	Majordomo@network.com
Message should contain a line stating…	subscribe hippi <*your E-mail address*>
Internet address for distribution via the HIPPI reflector	hippi@network.com

T11 E-mail Reflector (for T11 meeting notices, agendas etc.)

Internet address for subscription to the T11 reflector:	Majordomo@network.com
Message should contain a line stating…	subscribe T11 <*your E-mail address*>
Internet address for distribution via T11 reflector:	t11@network.com

Web sites:

HIPPI and ST Standards Activities	http://www.cic-5.lanl.gov/~det
T11 Activities	http://www.dpt.com/t11
NCITS	http://www.x3.org/

T11 Document Distribution :

Global Engineering
15 Inverness Way East
Englewood, CO 80112-5704
Voice : 303-792-2181 or 800-854-7179
FAX : 303-792-2192

PATENT STATEMENT

CAUTION: The developers of this standard have requested that holder's of patents that may be required for the implementation of the standard, disclose such patents to the publisher. However, neither the developers nor the publisher have undertaken a patent search in order to identify which, if any, patents may apply to this standard.

As of the date of publication of this standard and following calls for the identification of patents that may be required for the implementation of the standard, some such claims have been made. No further patent search is conducted by the developer or the publisher in respect to any standard it processes. No representation is made or implied that licenses are not required to avoid infringement in the use of this standard.

Comments on Rev 1.4

This is a preliminary document undergoing lots of changes. Many of the additions are just place holders, or are put there to stimulate discussion. Hence, do not assume that the items herein are correct, or final – everything is subject to change. This page tries to outline where we are; what has been discussed and semi-approved, and what has been added or changed recently and deserves your special attention. This summary relates to changes since the previous revision. Also, previous open issues are outlined with a single box, new open issues ones are marked with a double bar on the left edge of the box.

Changes are marked with margin bars so that changed paragraphs are easily found, and then highlights mark the specific changes. The list below just describes the major changes, for detail changes please compare this revision to the previous revision. **The major technical changes are printed in bold.**

Please help us in this development process by sending comments, corrections, and suggestions to the Technical Editor, Don Tolmie, of the Los Alamos National Laboratory, at det@lanl.gov. If you would like to address the whole group working on this document, send the comment(s) to hippi@network.com.

1. In 3.1.9, changed "…data movement function…" to "…data movement…".
2. In 3.1.18, changed "…identifying a data Destination's memory area…" to "…identifying an area of memory…".
3. In 3.1.20, changed "Eight bytes…" to "Six bytes…".
4. In 3.1.26, added a whole new definition for "RFC" with most of the text being taken from the previous footnote to 5.2.7.
5. In 3.3, added new acronyms for IP, RFC, and UDP.
6. In 4.2, 3rd paragraph, next to last sentence, changed "…expose multiple persistent…" to "…allocate multiple persistent…". (Did a global change so that most of the persistent memory is "allocated" and one-time buffers are "exposed".)
7. In 4.2, 4th paragraph, last sentence, changed "…Blocks are flow-controlled, and retransmitted when necessary…" to "…Blocks for transmission and retransmission are enabled (flow-controlled)…".
8. In 4.2, 5th paragraph (just below Figure 3), changed "The persistent memory region is equivalent to a single-Block Transfer and the data unit exchanged, without explicit flow control, is a single Block." to "A persistent memory region is similar to a region of memory allocated for the transfer of a Block (i.e., a starting Bufx and Offset), but is different in that the persistent memory region can be used for multiple Put, Get, or FetchOP sequences. The data unit exchanged in a Put, Get, or FetchOP sequence, without explicit flow control, is a Block (but the Block may be smaller than the persistent memory region).".
9. In 4.2, 6th paragraph changed the last sentence with "…for any reason…" to "…for any purpose…".
10. In 4.2, 7th paragraph (just below Figure 4), changed "…shows the model…" to "…shows a model…", changed "…Destination side." to "…Destination side, data structures.", changed "…would be similar." to "…may be similar.". In the second bullet, changed "…for operation Schedule Headers…" to "…for Schedule Headers…".
11. In 4.2, next to last paragraph, changed "…includes pointers…" to "…included references (Mx_n)…", changed "The Transfer Descriptor for a persistent memory region contains the size of the region (T_len, in bytes), and indexes (i.e., Mx) to Block Descriptors." to "There is only one Block Descriptor for a persistent memory region.", and changed "…Descriptors (one for each persistent memory region)…" to "…Descriptors…".
12. In Figure 5, deleted "CCI" from the "Parameters for local end", changed "B_id_0, B_id_1, and B_id_n" to "Mx_0, Mx_1, and Mx_n", and added "NOTE – Additional parameters may be required for control of lower layers. (See annex A.)", changed the title from "Model of a local end device's Destination side" to "A Destination side data structure model".

13. **In 5.1, added the second sentence reading "An Op code (see 8.1), and a checksum (see 8.3), are part of every operation but are not included in this clause.".** Split the paragraph into two paragraphs.
14. In 5.1.1, under "Request_Connection", "Flags", changed "F flags…" to "F bits…", and changed "…supports out of order Block delivery…" to "…can send and receive Blocks in any order…".
15. In 5.1.1, under "Connection_Answer", changed "…for use by…" to "…for use with…". Under "Flags", changed "F flags…" to "F bits…", and changed "…supports out of order Block delivery…" to "…can send and receive Blocks in any order…".
16. **In 5.1.2, under "Request_Disconnect", changed "– R-Port, I-Port, and R-Key…" to "– R-Port, I-Port, R-Key, and I-Key…", i.e., added I-Key.**
17. **In 5.1.2, under "Disconnect_Answer"**, changed "…shall retain…" to "…shall retain (for at least twice the Op_timeout period)…". Added "This delay allows for lost or damaged teardown operations to be re-issued.". **Changed "– I-Port, R-Port, and I-Key…" to "– I-Port, R-Port, I-Key, and R-Key…", i.e., added R-Key.**
18. **In 5.1.2, under "Disconnect_Complete"**, changed "…two times the Op_timeout, both the Initiator and Responder…" to "…twice the Op_timeout period, the Initiator…". Added "Disconnect_Complete is the last step in the three-way teardown handshake.". **Changed "– R-Port, I-Port, and R-Key…" to "– R-Port, I-Port, R-Key, and I-Key…", i.e., added I-Key.**
19. In 5.2.3, added the sentence reading "Note that the buffer sizes in each direction may be different.". Split the paragraph into two, and changed "…sizes. Get…", to "…sizes, except that Get…".
20. In 5.2.4, changed "…size of a flow-controlled unit (i.e., a Block, see 6.2.4)." to "…size of a Block (see 6.2.4).". Changed "…size authorized in…" to "…size in…".
21. In 5.2.6, split the third paragraph into two, and added at the end of the first "A received Slot value of x'FFFF' indicates that the remote end does not implement Slot accounting.". In the next paragraph, changed "…x'FFFFFFFF'…" to "…x'FFFF'…", and changed "…not implement Slot accounting." to "…not implementing Slot accounting or cannot supply an update to the Slots value.". In the last paragraph, added the sentence reading "A received Slots value of x'FFFF' indicates that the remote end device cannot supply an update to the Slots value now.".
22. In 5.2.7, second paragraph, changed "…(see 5.3)." to "…(see annex A).".
23. **Deleted clause 5.3 titled "Connection control information (CCI)". Most of the text was moved to annex A.**
24. **In 6.1, added the sentence reading "An Op code (see 8.1), and a checksum (see 8.3), are part of every operation but are not included in this clause.".**
25. In 6.1.1.1, under "Request_State_ Response", changed "…may be issued by the Responder…" to "…shall be issued by the Responder in response to the Request_State operation above.
26. In 6.1.1.2, under "Request_State", changed "…shall be issued…" to "…may be issued…". Under "R-id", changed "…the Responder's…" to "…specifies the Responder's…". Added "– I-id specifies the Initiator's Transfer identifier (see 6.2.1).".
27. Did a global change to use consistent word for the different actions. For example, "assigns" means the end device can pick any value, "specifies" means that the end device picks one of several pre-assigned values or uses a specified algorithm, and "echoes" means that the end device uses a specific pre-assigned value.
28. **In 6.1.1.2, under Request_State_Response",** changed "…Responder, specifying…" to "…Responder in response to the Request_State operation above. In this Request_State Response, the Responder specifies…". **Added "– R-id echoes the Responder's Transfer identifier (see 6.2.1).".**

29. **In 6.1.1.3, under "Request_State"**, changed "…shall be issued…" to "…may be issued…". Changed "– R-id, the…" to "– R-id specifies the…". **Added "– I-id specifies the Initiator's Transfer identifier (see 6.2.1)."**.
30. **In 6.1.1.3, under "Request_State_Response**", changed "…Responder, specifying…" to "…Responder in response to either: the Request_State operation above, or to a Data operation with Send_State = 1. In this Request_State Response, the Responder specifies…". **Added "– R-id echoes the Responder's Transfer identifier (see 6.2.1).".**
31. In 6.1.1.4, under "End" changed "…shall be issued…" to "…may be issued…". Changed "– R-id echoes…" to "– R-id specifies…" and changed "…identifier to specify a specific Transfer…" to "…identifier…". Made the same change to the I-id.
32. In 6.1.2, changed "…must exist…" to "…shall exist…", and changed "…can be initiated" to "…is initiated".
33. In 6.1.2, under "Request_To_Send", changed "…shall be issued…" to "…may be issued…". Changed "D flags…" to "D bits…" (this is a global change that has a margin bar here but not for the other instances).
34. In 6.1.2, under "Request_Answer", changed "…the subsequent…" to "…the Request_To_Send has been accepted but the subsequent…".
35. In 6.1.2, under "Clear_To_Send", changed "– B_num assigns…" to "– B_num specifies…".
36. In 6.1.2, under "Data", changed "– STU_num assigns…" to "– STU_num specifies…".
37. In 6.1.2, under "End", added "(the Write sequence)"
38. In 6.1.3, made similar changes as were made in 6.1.2.
39. In 6.1.4, changed "…are preceded by a sequence that exposes…" to "…shall be preceded by a sequence that allocates…". Changed "…must exist before these sequences can be…" to "…shall exist before a persistent memory region allocation sequence is…". Split the paragraph into two, and changed "…exposed…" to "…allocated…" in two places.
40. In 6.1.4.1, under "Request_Memory_Region", changed "…shall be issued…" to "…may be issued", and changed "…expose…" to "…allocate…".
41. In 6.1.4.1, under "Request_Answer", changed "…the subsequent…" to "…the Request_Memory_Region has been accepted but the subsequent…".
42. In 6.1.4.1, under "Memory_Region_Available", changed "…they must obey…" to "…by…".
43. In 6.1.4.2, under "Data", changed "– STU_num assigns…" to "– STU_num specifies…", and changed "– B_num specifies…" to "– B_num assigns…".
44. In 6.1.4.2, under "End", changed "…abort the Put…" to "…abort the Put sequence…".
45. **In 6.1.4.3, under "Get"**, changed "…shall be issued…" to "…may be issued…". Changed "– R-Mx assigns…" to "– R-Mx echoes…". **Changed T_len from 32 bits to 16 bits.**
46. In 6.1.4.3, under "Request_Answer", changed "…the subsequent…" to "…the Get has been accepted but the subsequent…".
47. In 6.1.4.3, under "Data", changed "– STU_num assigns…" to "– STU_num specifies…".
48. In 6.1.4.3, under "End", changed "…abort the Get…" to "…abort the Get sequence…".
49. In 6.1.4.4, changed "…fetches from…" to "…fetches data from…".
50. **In 6.1.4.4, under "FetchOP", changed "…shall be issued…" to "…may be issued…". Changed "– B_num field = x'00000008'… to "– Op_len field = x'0008'…" and changed "…32-bit length…" to "…16-bit length…".**
51. **In 6.1.4.4., under "Request_Answer", changed "…T_len ≠ x'00000008'…" to "…T_len ≠ x'0008'…".** Under Flags, changed "…"Request_To_Send has been rejected…" to "…FetchOP has been rejected…", and changed "…the subsequent…" to "…the FetchOP has been accepted but the subsequent…".

52. In 6.1.4.4, under "End", changed "…abort the FetchOP…" to "…abort the FetchOP sequence…".

53. In 6.2.1, under F-id and G-id, added "sequence" at the end of each paragraph. Under I-id and R-id, added "or persistent memory region" at the end of each paragraph. In the last paragraph, changed "…and only return…" to "…and shall only return…".

54. In 6.2.2, changed the first sentence to "Like the Ports, Keys, and Transfer identifiers, each end device shall also select its own 16-bit Memory Index (Mx). The Mx parameter provides a mechanism to identify an area of memory associated with a Block or persistent memory region.". Changed "…associated Data operations shall echo the Mx…" to "…Data operations on this memory region shall use the assigned Mx…".

55. In 6.2.3, split out the text into three separate paragraphs, one for 64-bit T_len with Read and Write sequences, one for 64-bit T_len for setting up a persistent memory region, and **one for 16-bit T_len for Get and FetchOP sequences**.

56. In 6.2.4, split the first paragraph into two paragraphs. In the second paragraph, added "of a Request Block state" to say which we are talking about. In the third paragraph, changed "Blocks comprising a Transfer shall…" to "Blocks shall…", and deleted "otherwise go-back N retransmission must be used.".

57. In 6.2.6, added "but STU sizes shall be no larger than specified in 5.2.4" at the end of the paragraph.

58. In 6.2.7, added "(see annex B)" at the end of the third paragraph.

59. **In 6.2.9, changed the size of the opaque data from 8 bytes to 6 bytes, and changed its location from the Bufx_2 field to the Op_len field.**

60. In 6.2.11, changed "Blocks" to "memory region" in two places, and changed "exposed" to "allocated".

61. In 6.3 towards the end, changed "Max_STU size" to "Max_STU" in two places.

62. **In Table 1, added Get and FetchOP as Rejected operations, and Request_Answer as the Response to each.**

63. Removed old clause "7.4 Lost Operations" and moved most of the text to 10.1 and 10.2.

64. **In figure 12, split the Bufx_2 field into two 16-bit fields; Cksum on the left and Op_len on the right.**

65. **In 8.3, added initial text for a checksum based on the TCP/IP checksum.**

66. **In 9, added "shall not be checked at the receiver" to the next to last bullet.**

67. **In Tables 3, 4, 5, 6, and 7, split the Bufx_2 field into two fields; Cksum and Op_len. Cksum is the parameter for every operation.**

68. **In Table 3, moved I-Max_Block and R-Max_Block from the old Bufx_2 field to the Offset_2 field. Added I-Key, R-Key, and I-Key in the Offset field for the teardown sequence.**

69. **In Tables 5, 6, and 7, changed the Opaque data from 64 bits to 32 bits, i.e., removed it from the new Cksum field.**

70. **In Table 7, moved the I-Bufx parameter in the Get and FetchOP operations to the B_num field, and the length to the Op_len field.**

71. **In 10.1, added the third bullet about the maximum queuing delay. The text in the second full paragraph was extracted from old 7.4.**

72. In 10.2, most of the first paragraph is text extracted from the old 7.4.

73. In Table 8, the title was changed with the addition of "with mandatory retry". Added "or Request_Answer" as a response to Get. Added "Data with Send_State=1" with "Request_Answer" as a response. Reordered the items into alphabetical order.

74. **In 10.4, added new text about checksum errors.**

75. In 10.5.1, 10.5.2, and 10.6.3, changed "…logged…" to "…shall be logged…".

76. In 10.6.1, added text about Disconnect operations possibly not having valid Port and Key values, and generating legal responses when an invalid Disconnect operation is received.
77. **In 10.7.1, changed the title from "Invalid S-id" to "Invalid Transfer identifier". Changed the text so that only the destination identifier is checked and the S-id field is not checked. Changed the error logged from "Invalid_S-id_Error" to "Invalid_D-id_Error".**
78. **In 10.7.2, added an new clause about checking the Mx parameter.**
79. In 10.7.3, added the other operations that specify Data channels.
80. In 10.7.4, reworded the text about out of range Block numbers, turning it into a list with two bullets. Added Get and FetchOP to the checking for Bufx and Offset out of range.
81. In 10.7.5, added "or persistent memory region" in two places.
82. **In 10.7.6, deleted "Request_To_Send_Response". Changed "…should discard…" to "…shall discard…".**
83. In 10.7.7, changed "…flag = 1 and use of that flag…" to "…flag value that…".
84. In 10.7.8, changed "…from systems that are capable of retransmission." to "…from a data Source that supports Out_Of_Order (see 5.1.1 and 8.2)."
85. **In Table 9, added "Cksum_Error, all", and "Invalid_Mx_Error, Data". Changed "Invalid_S-id_Error" to "Invalid_D-id_Error" and "all with an S_id" to "all with a non-zero D_id".** Reordered the items in the table to be in alphabetical order.
86. In Annex A, the text in the second and following paragraphs came from the old 5.3. In the third paragraph, changed "…is passed from the ULP to the specified LLP and stored…" to "…is passed to the specified LLP and may be stored…". In the fourth paragraph, added the sentence reading "ST does not provide the initial CCI; it may come from the ULP or from another protocol.", and changed "…should retain…" to "…would typically retain…".
87. **In figures A1, A2, and A3, changed the Bufx_2 field to Cksum and Op_len.**

Contents

Tables

Figures

Annexes

Foreword (This foreword is not part of American National Standard X3.xxx-199x.)

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

This document includes annexes which are informative and are not considered part of the standard.

Requests for interpretation, suggestions for improvement or addenda, or defect reports are welcome. They should be sent to the National Committee for Information Technology Standards, 1250 Eye Street, NW, Suite 200, Washington, DC 20005.

This standard was processed and approved for submittal to ANSI by NCITS. Committee approval of the standard does not necessarily imply that all committee members voted for approval. At the time it approved this standard, the NCITS had the following members:

(List of NCITS members to be included in the published standard by the ANSI Editor.)

Technical Committee T11 on Device Level Interfaces, which reviewed this standard, had the following participants:

(List of T11 Committee members, and other active participants, at the time the document is forwarded for public review, will be included by the Technical Editor.)

Task Group T11.1 on the High-Performance Parallel Interface, which developed this standard, had the following participants:

(List of T11.1 Task Group members, and active participants, at the time of document is forwarded for public review will be included by the Technical Editor.)

Introduction

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

Characteristics of a ST include:

– A hierarchy of data units (Scheduled Transfer Units (STUs), Blocks, and Transfers).

– Support for flow-controlled Block Read and Write sequences.

– Support for single Block Get and Put sequences.

– Support for Fetch and modify sequences.

– Parameters exchanged between end devices for Port selection, transfer identification, and operation validation.

– Features supporting efficient mapping between the issuer's and receiver's natural buffer sizes.

– Features supporting Block striping.

– Provisions for resending partial Transfers for error recovery.

– Mappings onto HIPPI-6400-PH, HIPPI-FP (for HIPPI-800 traffic), Fibre Channel, and Ethernet lower-layer protocols.

American National Standard for Information Technology –

Scheduled Transfer Protocol (ST)

1 Scope

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

Specifications are included for:

– Virtual Connection setup and teardown;

– determining the number of operations the other end can accept;

– determining the buffer size of the other end;

– exchanging Key, Port, transfer identifiers, and buffer size values specific to the end nodes;

– determining a maximum size transmission unit that will not overrun receiver buffer boundaries;

– acknowledging partial transfers so that buffers can be reused;

– providing means for resending partial Transfers for error recovery; and

– terminating transfers in progress.

Note that some Scheduled Transfer Protocol implementations work best with in-order delivery by the LLP, which may not be available on all media.

2 Normative references

The following standards contain provisions which, through reference in the text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

Copies of the following documents can be obtained from ANSI: Approved ANSI standards, approved and draft international and regional standards (ISO, IEC, CEN/CENELEC, ITUT), and approved and draft foreign standards (including BSI, JIS, and DIN). For further information, contact ANSI Customer Service Department at 212-642-4900 (phone), 212-302-1286 (fax) or via the World Wide Web at http://www.ansi.org. Additional availability contact information is provided below as needed.

2.1 Approved references

ANSI X3.183-1991, *High-Performance Parallel Interface – Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH)*

ANSI X3.210-1992, *High-Performance Parallel Interface – Framing Protocol (HIPPI-FP)*

ANSI/IEEE Std 802-1990, *IEEE Standards for Local and Metropolitan Area Networks: Overview and architecture (formerly known as IEEE Std 802.1A, Project 802: Local and Metropolitan Area Network Standard — Overview and Architecture).*

ISO/IEC 8802-2:1989 (ANSI/IEEE Std 802.2-1989), *Information Processing Systems – Local Area Networks – Part 2: Logical link control.*

2.2 References under development

At the time of publication, the following referenced standards were still under development. For information on the current status of the document, or regarding availability, contact the relevant standards body or other organization as indicated. For information about obtaining copies of this document or for more information on the current status of the document, contact National Committee for Information Technology Standards, 1250 Eye Street, NW, Suite 200, Washington, DC 20005, 202-626-5746.

ANSI X3.xxx-199x, *High-Performance Parallel Interface – 6400 Mbit/s Physical Layer (HIPPI-6400-PH)*

3 Definitions and conventions

3.1 Definitions

For the purposes of this standard, the following definitions apply.

3.1.1 atomic: An indivisible operation or transaction, i.e. it updates all of its defined state variables before another operation can take place on the same variables.

3.1.2 Block: An ordered set of one or more STUs within a Scheduled Transfer. (See figure 3 and 6.2.4.)

3.1.3 Buffer Index (Bufx): A 32-bit parameter identifying the starting address of a data buffer. (See 6.2.7.)

3.1.4 Bufsize: An end device's buffer size. (See 5.2.3.)

3.1.5 connection control information (CCI): Media-dependent information, e.g., physical-layer addresses, passed from the ULP for use by the physical layer below ST.

3.1.6 Control Channel: The logical channel that carries the Control operations.

3.1.7 Control operation: A control function consisting of a Schedule Header and an optional 32-byte payload. (See figure 4.)

3.1.8 Data Channel: The logical channel that carries the data payload.

3.1.9 Data operation: A data movement consisting of a Schedule Header and up to 2 gigabytes of user payload. (See figure 4).

3.1.10 Destination: The end device that receives an operation or data.

3.1.11 Get: An operation to read data from a persistent memory region on a remote end device. (See 6.1.4.3.)

3.1.12 FetchOP: An atomic operation to read data from a persistent memory region on a remote end device and execute some function on the persistent memory location, e.g., increment. (See 6.1.4.4.)

3.1.13 Initiator: The end device that starts a sequence of operations. This is typically a host computer system, but may also be a non-transparent translator, bridge, or router.

3.1.14 intermediate device: A non-transparent device (e.g., translator, bridge, or router), between the end device that generates the data payload and the end device that receives and operates on the data payload.

3.1.15 Key: A local identifier used to select and validate operations. (See 5.2.2.)

3.1.16 log: The act of making a record of an event for later use.

3.1.17 lower-layer protocol (LLP): A protocol below the Scheduled Transfer Protocol, e.g., a physical layer.

3.1.18 Memory Index (Mx): A parameter identifying an area of memory. (See 6.2.2).

3.1.19 Offset: A parameter specifying the data's starting point relative to the start of a Bufx. (See 6.2.7.)

3.1.20 Opaque data: Six bytes of Source ULP to Destination ULP peer-to-peer information carried in the Scheduled Header separately from the data payload. (See 6.2.9)

3.1.21 operation: The procedure defined by the parameters in a Schedule Header, and any payload associated with that Schedule Header (see figure 4). The code in the Schedule Header's "Op" field identifies the operation's name/function (see table 2).

3.1.22 optional: Characteristics that are not required by ST. However, if any optional characteristic is implemented, it shall be implemented as defined in ST.

3.1.23 persistent: Memory that is maintained for multiple Put, Get, and FetchOP operations. (See 6.1.4 and 6.2.11.)

3.1.24 Port: A logical connection within an end device. (See 5.2.1.)

3.1.25 Put: An operation to write data into a persistent memory region on a remote end device. (See 6.1.4.2.)

3.1.26 Request For Comment (RFC): RFC (Request For Comment) documents are working standards documents from the TCP/IP internetworking community. Copies of these documents are available from numerous electronic sources (e.g., http://www.ietf.org) or by writing to Internet Engineering Task Force (IETF) Secretariat, c/o Corporation for National Research Initiatives, 1895 Preston White Drive, Suite 100 Reston, VA 20191-5434, USA.

3.1.27 Responder: The end device that responds to the sequence of operations started by the Initiator. This is typically a host computer system, but may also be a non-transparent translator, bridge, or router.

3.1.28 Scheduled Transfer: An information transfer, normally used for bulk data movement, where the end devices prearrange the transfer using the protocol defined in this standard.

3.1.29 Scheduled Transfer Unit (STU): The data payload portion of a Data operation. STUs are the basic components of Blocks and are the smallest units transferred. (See figure 3 and 6.2.6.)

3.1.30 sequence: An ordered group of operations providing a particular function, e.g., Read, Write, Get, etc., between an Initiator and a Responder. The roles of Initiator and Responder are constant for all operations in the sequence.

3.1.31 Slot: A space reserved for a Control operation, or the Schedule Header portion of an STU, in the end device. (See 5.2.6.)

3.1.32 Source: The end device that sends an operation or data.

3.1.33 Sync: A parameter used to synchronize the state across a Virtual Connection. (See 5.2.6.)

3.1.34 Transfer: An ordered set of one or more Blocks within a Scheduled Transfer. (See figure 3 and 4.2.)

3.1.35 Universal LAN MAC address (ULA): A logical address that uniquely identifies a Source or Destination. The ULA conforms to the 48-bit MAC address specified by the IEEE 802 Overview Standard.

3.1.36 upper-layer protocol (ULP): The protocol above ST. A ULP could be implemented in hardware or software, or could be distributed between the two.

3.1.37 Virtual Connection: A bi-directional logical connection used for Scheduled Transfers between two end devices. A Virtual Connection contains a logical Control Channel and one or more logical Data Channels in each direction. (See 5.)

3.2 Editorial conventions

A number of conditions, sequence parameters, events, states, or similar terms are printed with the first letter of each word in uppercase and the rest lowercase (e.g., Block, Transfer). Any lowercase uses of these words have the normal technical English meaning.

The word *shall,* when used in this American National standard, states a mandatory rule or requirement. The word *should,* when used in this standard, states a recommendation.

Multiword parameters and field names are joined with an underscore, e.g., D_Port. A parameter associated with a particular end device uses a single letter prefix and a hyphen as a joiner, e.g., I-Port denoting the Initiator's Port.

All numbers are represented as unsigned integers.

3.2.1 Binary notation

Binary notation is used to represent relatively short fields. For example a two-bit field containing the binary value of 10 is shown in binary format as b'10'. An "x" in a bit position indicates a "don't care" value.

3.2.2 Hexadecimal notation

Hexadecimal notation is used to represent some fields. For example a two-byte field containing a binary value of b'1100010000000011' is shown in hexadecimal format as x'C403'.

3.3 Acronyms and other abbreviations

Ack	acknowledge indication
CCI	connection control information
D_	associated with the Destination
DSAP	Destination Service Access Protocol
FTP	File Transfer Protocol
HIPPI	High-Performance Parallel Interface
I-	Prefix for an Initiator's parameter
id	identifier
IP	Internet Protocol
IEEE	Institute of Electrical and Electronic Engineers
LAN	local area network
LLP	lower-layer protocol
MAC	Media Access Control
R-	Prefix for a Responder's parameter
RFC	Request For Comment
S_	associated with the Source
SNAP	SubNetwork Access Protocol
SSAP	Source Service Access Protocol
ST	Scheduled Transfer Protocol
STU	Scheduled Transfer Unit
TCP	Transmission Control Protocol
UDP	User Data Protocol
ULA	Universal LAN address
ULP	upper-layer protocol

4 System overview

This clause provides an overview of the structure, concepts, and mechanisms used in Scheduled Transfers. Figure 1 gives an example of Scheduled Transfers being used to communicate between a local end device and a remote end device over some physical media. Annex C describes the steps in a typical Scheduled Transfer. Figure 2 shows ST being used over different media.

4.1 Control Channels and Data Channels

Each end device shall have a Control Channel and one or more Data Channels. Control operations shall be exchanged over the Control Channel. Scheduled Transfer Units (STUs), i.e., data payload, shall be exchanged over the Data Channel(s). The information volume on the Data Channel(s) will probably be many times the volume on the Control Channel; hence the available bandwidths should be balanced accordingly. For best performance, the Control Channel should have low latency.

4.2 System model

Scheduled Transfers between end devices are pre-arranged to decrease computational overhead during the Transfer by allocating buffers at each end device. The bi-directional path between the end devices is called a Virtual Connection. The end devices can be either Initiators or Responders. The Initiator is the end device that starts a Read, Write, request persistent memory region, Put, Get, or FetchOP sequence. The Responder is at the other end. The end device sending an operation is the Source and the end device receiving the operation is the Destination. The end device sending data is the data Source and the end device receiving the data is the data Destination.

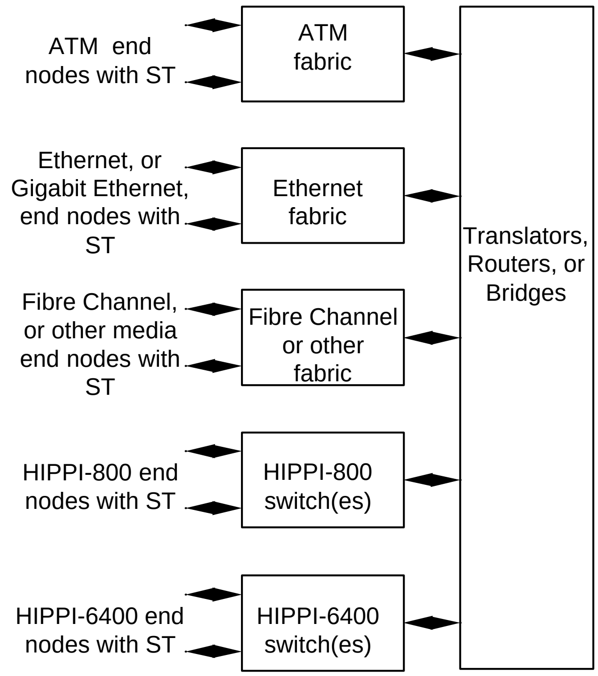


Figure 2 – ST over different media

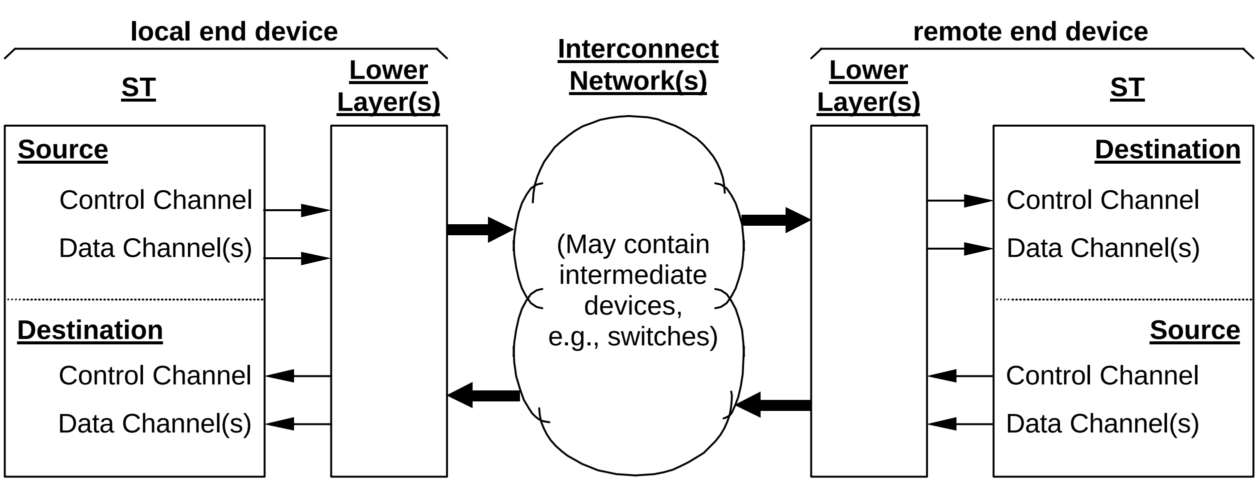


Figure 1 – System overview

Once the data Destination has indicated its ability to accept a Block, the Virtual Connection should not become congested. In essence, the data Destination smoothly controls the flow. For comparison, without pre-arranging the buffers, the data Source would blindly send data into the interconnection network where it might have to wait for buffers to be assigned in the data Destination. On the down-side, Scheduled Transfers require additional Control operations and round-trip latency. Once established, a Virtual Connection may be used to carry multiple Read and Write Transfers, allocate multiple persistent memory regions, or execute multiple Put, Get, and FetchOP sequences. This Scheduled Transfer protocol does not handle network resource reservations.

Multiple independent Write or Read sequences may be executed to move user data units, called Transfers, over Virtual Connections. As shown in figure 3, a Transfer is composed of one or more Blocks, and Blocks are composed of one or more STUs. The Scheduled Transfer protocol shall package the Transfer in Blocks and STUs for delivery using a lower layer protocol (LLP) and media. The Blocks for transmission and retransmission are enabled (flow-controlled), with Clear_To_Send operations.

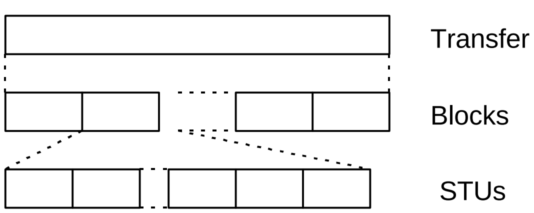


Figure 3 – User data hierarchy

Multiple Put, Get, and FetchOP sequences may also be executed to move user data to/from a persistent memory region. A persistent memory region is similar to a region of memory exposed for the transfer of a Block (i.e., a starting Bufx and Offset), but is different in that the persistent memory region can be used for multiple Put, Get, or FetchOP sequences. The data unit exchanged in a Put, Get, or FetchOP sequence, without explicit flow control, is a Block (but the Block may be smaller than the persistent memory region).

As shown in figure 4, an STU shall be the data payload portion of a Data operation. A Data operation shall consist of a 40-byte Schedule Header and an STU of up to 2 gigabytes (2^{31} bytes). A Control operation shall consist of a 40-byte Schedule Header, and may contain an additional 32 bytes of optional payload. The optional payload can be used by a ULP entity for any purpose, e.g., passing file names.

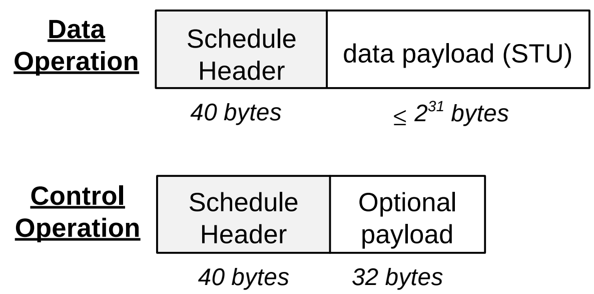


Figure 4 – Transmission units

Figure 5 shows a model of a local end device's, Destination side, data structures. An end device's Source side may be similar.

As Control operations and Data operations are received, the Schedule Header of each is placed in the Schedule Header queue for execution. State information about the number of empty Slots in the queue is available to the other end so that it can avoid overrunning the queue.

The Virtual Connection Descriptor (selected and validated by the ordered set remote-Port, local-Port, and local-Key), contains:

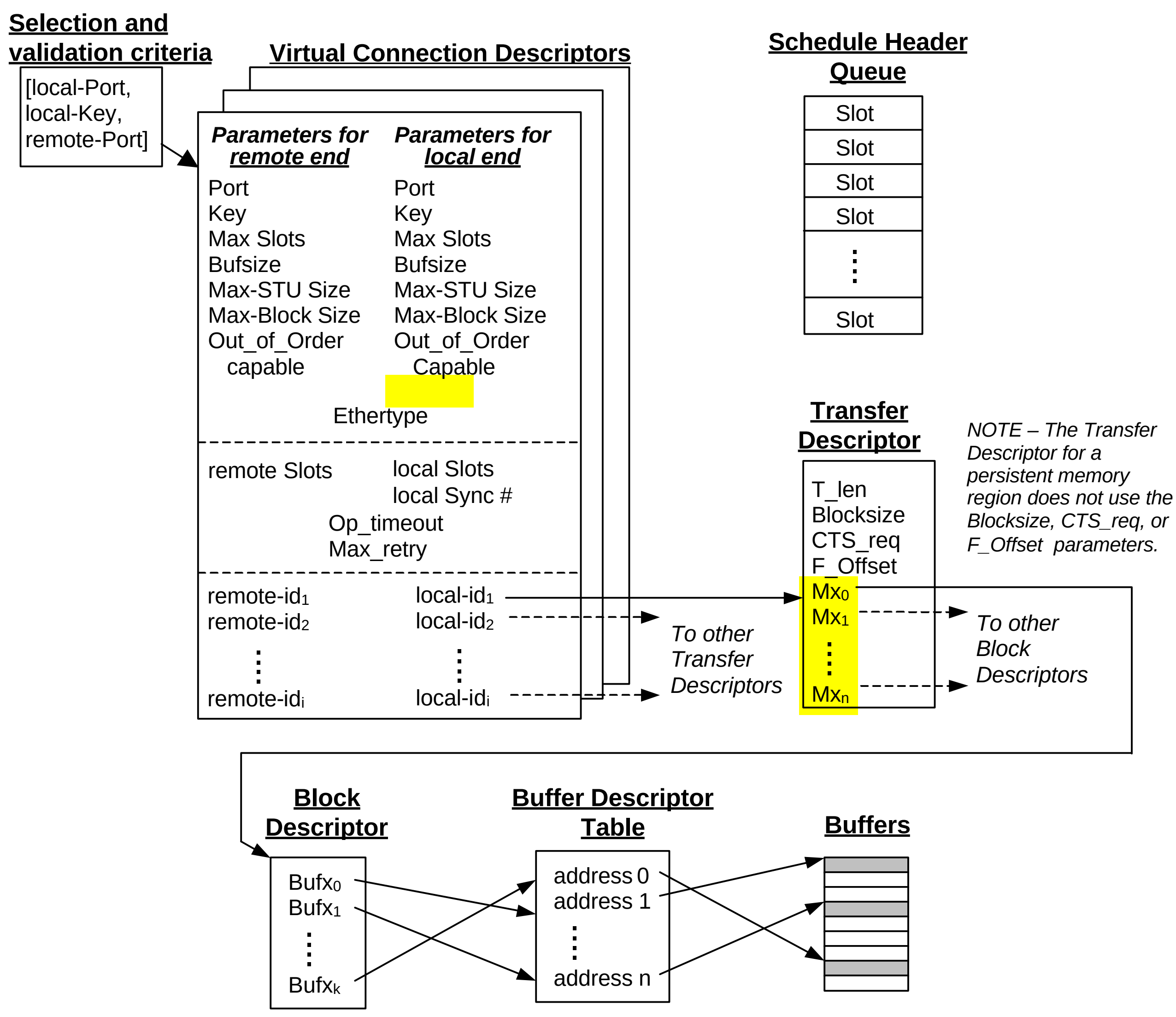
– static parameters identifying the Virtual Connection from the view of both the remote end device and local end device (the top portion of the Virtual Connection Descriptor box in figure 5). Since the values assigned to local-Port and local-Key are determined by the local end device, it is left up to the local end device as to which parameters to use to select and which to use to validate;

– current state information about the number of empty "Slots" for Schedule Headers associated with this Virtual Connection, and Retry and Timeout parameters;

– identifiers for each of the Virtual Connection's Transfers or persistent memory regions.

A Transfer Descriptor, for each Transfer, contains the Transfer length (T_len, in bytes), the Block size (in bytes), and includes references (Mx_n), to Block Descriptors. There is only one Block Descriptor for a persistent memory region. The Block Descriptors identify the set of contiguous Buffer Index (Bufx) values assigned to the Block or persistent memory region. And finally, the Buffer Descriptor Table provides a base memory address for each Bufx.

In an effort to achieve maximum transfer rates and efficiency, the receiver's job is made as easy as possible, even at the expense of the transmit side. It is expected that after validating an operation in the receiving end, only a single lookup will be needed to derive the absolute memory address and correctly place the data.



NOTE – Additional parameters may be required for control of lower layers. (See annex A.)

Figure 5 – A Destination side data structure model

5 Connection management

5.1 Connection management sequences

The connection management sequences are shown in figure 6 and detailed in table 3. An Op code (see 8.1), and a checksum (see 8.3), are part of every operation but are not included in this clause. Parameters transmitted as zeros, and not checked at the receiver, are marked in the tables as *, and are not included in this clause.

The end device that starts a sequence is called the Initiator, and the other end device is called the Responder. The label of an end device as Initiator or Responder remains constant within a sequence. An "I-" prefix indicates that a parameter is associated with the Initiator. An "R-" prefix indicates that a parameter is associated with the Responder.

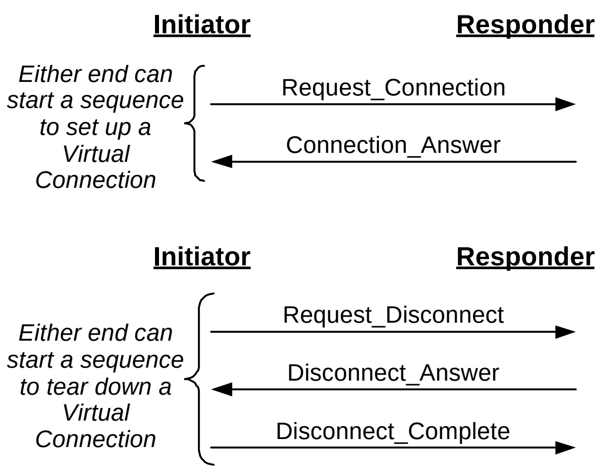


Figure 6 – Connection management example

5.1.1 Virtual Connection setup

A sequence consisting of a Request_Connection operation and a Connection_Answer operation shall be used to construct a symmetric Virtual Connection between two end devices. Either end device can initiate the Virtual Connection setup sequence, and sequences that follow need not use the same Initiator and Responder. (See table 3, C1.)

Request_Connection shall be issued by the Initiator to establish a Virtual Connection. Parameters are passed to inform the remote end of the Initiator's capabilities and preferences.

– Flags (see 8.2): F bits specify the Initiator's support for persistent operations and big/little endian ULP architecture. I = 1 specifies that an interrupt shall be generated at the Responder. O = 1 specifies that the Initiator can send and receive Blocks in any order (see 6.2.5).

– I-Slots specifies the initial number of Slots available in the Initiator for this Virtual Connection (see 5.2.6).

– R-Port specifies a "well-known Port" or other value to select an upper-layer protocol or service at the Responder (see 5.2.1).

– I-Port assigns the Initiator's Port value (see 5.2.1).

– I-Bufsize specifies the Initiator's buffer size (see 5.2.3).

– I-Key assigns the Initiator's Key (see 5.2.2).

– I-Max_STU specifies the maximum size STU the Initiator is prepared to receive (see 5.2.4).

– EtherType identifies the protocol associated with this Virtual Connection (see 5.2.7).

– I-Max_Block specifies the maximum Block size that the Initiator will send (see 5.2.5).

Connection_Answer shall be issued by the Responder upon receipt of a Request_Connection. The Responder may either reject the request, or reply with parameters to inform the Initiator of the Responder's capabilities and preferences. If accepted, a Virtual Connection will have been established for use with subsequent sequences.

– Flags (see 8.2): F bits specify the Responder's support for persistent operations and big/little endian ULP architecture. I = 1 specifies that an interrupt shall be generated at the Initiator. O = 1 specifies that the Responder can send and receive Blocks in any order (see 6.2.5). R = 1 specifies that the Responder has rejected this Virtual Connection.

– R-Slots specifies the number of Slots available in the Responder for this Virtual Connection (see 5.2.6).

– I-Port echoes the Initiator's Port value (see 5.2.1).

– R-Port assigns the Responder's Port value, which may not be the same as the R-Port value in the Request_Connection (see 5.2.1).

– I-Key echoes the Initiator's Key (see 5.2.2).

– R-Bufsize specifies the Responder's buffer size (see 5.2.3).

– R-Key assigns the Responder's Key (see 5.2.2).

– R-Max_STU specifies the maximum size STU the Responder is prepared to receive (see 5.2.4).

– R-Max_Block specifies the maximum Block size that the Responder will send (see 5.2.5).

The parameters assigned and specified during setup shall apply for the life of the Virtual Connection. Once established, the Virtual Connection is selected and validated as shown in figure 5 by the ordered set "remote-Port", "local-Port", and "local-Key".

5.1.2 Virtual Connection teardown

Either end device of a Virtual Connection can initiate a three-way handshake sequence to disconnect, or tear down, the Virtual Connection. (See table 3, C2.)

Request_Disconnect shall be issued by the Initiator to tear down a Virtual Connection and release the resources assigned to the Virtual Connection. A Request_Disconnect should only be issued when the Transfers are complete or appear to be stalled. Request_Disconnect is the first step in the three-way teardown handshake that decreases timeout dependency for releasing resources.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, R-Key, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

Disconnect_Answer shall be issued by the Responder to acknowledge receipt of a Request_Disconnect. The Disconnect_Answer issuer may release any buffers associated with this Virtual Connection, but shall retain (for at least twice the Op_timeout period), the Port and Key values for use in further Disconnect operations. This delay allows for lost or damaged teardown operations to be re-issued. Disconnect_Answer is the second step in the three-way teardown handshake.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– I-Port, R-Port, I-Key, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

Disconnect_Complete shall be issued by the Initiator to complete the three-way teardown handshake, acknowledging that the actions associated with a Request_Disconnect have been completed. After waiting an interval of at least twice the Op_timeout period, the Initiator shall release the Virtual Connection's Port and Key values. This delay allows for lost or damaged teardown operations to be re-issued. Disconnect_Complete is the last step in the three-way teardown handshake.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, R-Key, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

5.2 Connection management parameters

5.2.1 Ports

Ports identify upper-layer entities within an end device. The Port values shall be assigned by the local end device and have no meaning on the remote end device. For example, when the local end device requests a Virtual Connection, the local end device shall select the value for local-Port and shall send it to remote end device in the Request_Connection operation. The remote end device shall store the value as remote-Port and shall return it to the local end device in every operation over this Virtual Connection. Likewise, the remote end device shall select its Port value.

An exception to this is necessary when Virtual Connections are set up. The Destination Port (D_Port) value in a Request_Connection operation shall be a "well-known Port" or other value that selects an upper layer protocol or service. The interpretation of Port numbers for establishing Virtual Connections shall be as defined by the Internet Assigned Numbers Authority (IANA) and as published at http://www.isi.edu/div7/infra/iana.html. For example, if an FTP application were to use ST rather than TCP, then it would send a Request_Connection operation to Port 20 (the FTP Port) at the remote end system by specifying Port 20 in the D_Port field. The subsequent Connection_Answer operation would return a potentially different Port number in the S_Port field to use in subsequent data transfer operations (see 6).

If the incoming local-Port, remote-Port, and local-Key values are not a valid combination, then the operation shall not be executed (see 10.6.1).

5.2.2 Keys

Like the Ports (see 5.2.1), each end device shall select its own 32-bit Key for use on the Virtual Connection. The Key shall be assigned by the local end device and have no meaning on the remote end device.

If the incoming local-Port, remote-Port, and local-Key values are not a valid combination, then the operation shall not be executed (see 10.6.1).

5.2.3 Buffer size (Bufsize)

Each end shall define its receiving buffer size, in bytes. Buffer sizes may be the same as host page sizes. The buffer sizes shall be ≥ 256 bytes and shall be an integral power of two, i.e., $2^{Bufsize}$ where $8 \leq Bufsize \leq 63$. Note that the buffer sizes in each direction may be different.

Transmitting buffer sizes are not exchanged, and may be different from the receiving buffer sizes, except that Get and FetchOP operations require that the transmit and receive buffers be the same size (see 6.2.11).

5.2.4 Max_STU size

The Max_STU size, exchanged during Virtual Connection setup, establishes the maximum data payload size of an STU (see 6.3). Each end device declares the Max_STU size it is prepared to receive. The Max_STU size shall be no larger than its Bufsize. Intermediate devices with smaller buffer sizes may lower this value. Note that the Max_STU size in each direction may be different.

Additionally, an STU's maximum data payload size shall be ≥ 256 bytes and an integral power of two i.e., 2^{Max_STU} where $8 \leq Max\_STU \leq 31$.

5.2.5 Maximum Block size (Max_Block)

The maximum Block size, exchanged during Virtual Connection setup, establishes the maximum size of a Block (see 6.2.4). This number is the Max_Block parameter and is expressed as a power of two, i.e., 2^{Max_Block} where $8 \leq Max\_Block \leq 63$. Each end device declares the desired maximum Block size (see 6.2.5), it would like to send. Intermediate devices with smaller buffer sizes may lower the Max_Block value. End devices shall restrict the Block size in subsequent Clear_To_Send operations to ≤ 2^{Max_Block}. Note that the maximum Block size in each direction may be different.

5.2.6 Slots

The term Slot denotes memory at a Destination, associated with a specific Virtual Connection, reserved for storing the Schedule Header of an incoming operation. Each operation arriving at a Destination consumes one Slot, except Request_Connection operations (see 5.1.1), or Data operations which have Silent = 1 (see 8.2). A Source shall control the flow of operations by sending no more operations than there are Slots available at the Destination for this Virtual Connection. Any operations that are sent in excess of the number of available Slots may be discarded by the Destination (see 10.6.2).

In order to avoid potential deadlocks that can happen if a Source consumes all of its allocated slots at the Destination, the Source shall never consume all of its slots with data movement operations. Instead, the Source shall hold at least one Slot in reserve for possible use for an End, Request_State, Request_State_Response, or Request_Disconnect sequence.

An end device learns the initial number of Slots available (Slots value) at the remote end device during the Virtual Connection setup (see 5.1.1). A received Slots value of x'FFFF' indicates that the remote end does not implement Slot accounting.

Later, an end device obtains the current Slots value for a specific Virtual Connection by reading the Slots parameter in a received Request_State_Response. An end device may solicit a Request_State_Response from the remote end by one of two methods: by setting the Send_State flag in the Schedule Header of a Data operation, or by sending a Request_State operation. A received Slots value of x'FFFF' indicates that the remote end device does not implement Slot accounting or cannot supply an update to the Slots value.

NOTE – Slot accounting may not be needed when the maximum number of Control operations is otherwise bounded or where dropped operations are acceptable.

The received Slot value is a snapshot of the number of Slots available at the remote end device for the specified Virtual Connection when the remote end device received the soliciting operation. A received Slots value of x'FFFF' indicates that the remote end device cannot supply an update to the Slots value now. The local end device may continue to send operations after soliciting a Request_State_Response and may also solicit multiple responses before receiving a reply. The lower bound on the number of available Slots at the remote end device is determined by the local end device which adjusts its vision of the number of Slots to account for outstanding operations. The adjustment consists of subtracting the number of Slot-consuming operations sent by the local end device from the number of Slots indicated in the received Request_State_Response operation after a Request_State_Response solicitation.

5.2.7 EtherType

EtherType parameter values shall be as assigned in the current "Assigned Numbers" RFC, e.g., RFC 1700 (see http://www.iana.org/iana/). The EtherType parameter value in a Request_Connection operation shall identify the protocol associated with this ST Virtual Connection. For example, if ST is used to encapsulate TCP/IP, then this EtherType would be x'0800'. If ST is being used to encapsulate legacy HIPPI-FP data, then this EtherType would be x'8180'. EtherType = x'0000' means no further encapsulation.

The other use of EtherType is in the IEEE 802.2 LLC/SNAP header of an LLP, where EtherType = x'8181' specifies that the LLP protocol is carrying ST information. This EtherType parameter may be supplied to an LLP as part of the CCI (see annex A). For an example, see HIPPI-6400-PH.

6 Data movement

6.1 Data movement sequences

The data movement sequences are detailed in tables 4–7. An Op code (see 8.1), and a checksum (see 8.3), are part of every operation but are not included in this clause. Parameters transmitted as zeros, and not checked at the receiver, are marked in the tables as *, and are not included in this clause.

The end device that starts a sequence is called the Initiator, and the other end device is called the Responder. The label of an end device as Initiator or Responder remains constant within a sequence. An "I-" prefix indicates that a parameter is associated with the Initiator. An "R-" prefix indicates that a parameter is associated with the Responder.

The data movement sequences rely on having a Virtual Connection established (see 5).

6.1.1 Common sequences

The Request_State and End sequences, as shown in the figure 7 examples, and detailed in table 4, are available for use with other data movement sequences.

Request_State sequences are used to find one or more pieces of remote end information:

– Slot state – number of available Slots for a specific Virtual Connection (see 5.2.6);

– Transfer state – same as Slot state, plus the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly (see 6.2.4);

– Block state – same as Transfer state, plus whether the specified Block was received correctly (see 6.2.4).

End sequences are used to abort a Transfer currently in progress, or terminate a persistent memory region, from either end device.

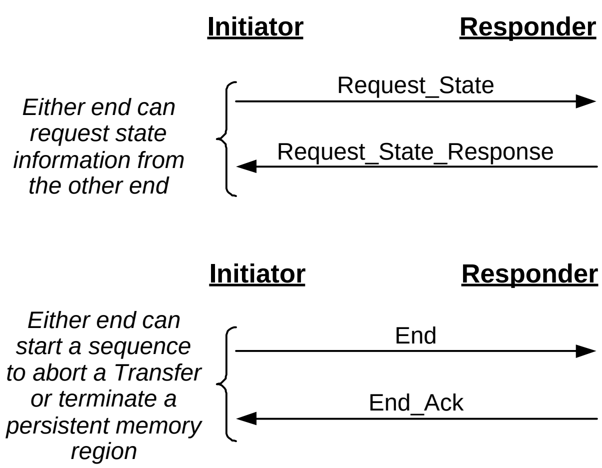


Figure 7 – Common sequence examples

6.1.1.1 Request Slot state

Request Slot state may be initiated by either end device to determine the number of available Slots at the other end device for this Virtual Connection. (See table 4, Com1.)

Request_State may be issued by the Initiator to request the Responder's view of its current number of available Slots in the Responder for this Virtual Connection.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– D_id field = x'FFFF', i.e., we are only requesting Slot accounting information (see 6.2.1).

– Sync assigns the Initiator's Sync value (see 6.2.8).

Request_State_Response shall be issued by the Responder in response to the Request_State operation above. In this Request_State_Response, the Responder specifies its view of the number of currently available Slots in the Responder for this Virtual Connection.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-Slots specifies the Responder's number of currently available Slots for this Virtual Connection (see 5.2.6).

– D_id field echoes the D_id field (see 6.2.1).

– Sync echoes the Initiator's Sync value (see 6.2.8).

6.1.1.2 Request Transfer state

Request Transfer state may be initiated by the data Source to determine the general status of a Transfer and to get the number of Slots currently available at the data Destination for this Virtual Connection. (See table 4, Com2.)

Request_State may be issued by the Initiator to request the Responder's view of its current number of available Slots in the Responder for this Virtual Connection and the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id specifies the Responder's Transfer identifier, i.e., asks about a specific Transfer (see 6.2.1).

– I-id specifies the Initiator's Transfer identifier (see 6.2.1).

– Sync assigns the Initiator's Sync value (see 6.2.8).

– B_num field = x'FFFFFFFF', i.e., we are not asking about a specific Block (see 6.2.4).

Request_State_Response shall be issued by the Responder in response to the Request_State operation above. In this Request_State Response, the Responder specifies its view of the number of currently available Slots in the Responder for Schedule Headers associated with this Virtual Connection, and the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Slots specifies the Responder's number of currently available Slots for this Virtual Connection (see 5.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– B_seq specifies the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly (see 6.2.4).

– Sync echoes the Initiator's Sync value (see 6.2.8).

– B_num field = x'FFFFFFFF' (see 6.2.4).

6.1.1.3 Request Block state

Request Block state may be initiated by the data Source to determine if the specified Block was received correctly, the general status of the Transfer, and the number of Slots available at the data Destination for this Virtual Connection. (See table 4, Com3.)

Request_State may be issued by the Initiator to request the Responder's view of its current number of available Slots in the Responder for this Virtual Connection, the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly, and the status of a specific Block.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id specifies the Responder's Transfer identifier, i.e., asks about a specific Transfer (see 6.2.1).

– I-id specifies the Initiator's Transfer identifier (see 6.2.1).

– Sync assigns the Initiator's Sync value (see 6.2.8).

– B_num asks about a specific Block (see 6.2.4).

Request_State_Response shall be issued by the Responder in response to either: the Request_State operation above, or to a Data operation with Send_State = 1. In this Request_State Response, the Responder specifies its view of: the number of currently available Slots in the Responder for Schedule Headers for the specified Virtual Connection, the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly, and if the Block specified (B_num in the Request_State or Data operation being responded to), was received correctly.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Slots specifies the Responder's number of currently available Slots for this Virtual Connection (see 5.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– B_seq specifies the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly (see 6.2.4).

– Sync echoes the Initiator's Sync value (see 6.2.8).

– B_num indicates if the specified Block was received correctly (see 6.2.4).

6.1.1.4 End sequence

End sequences allow either end device of the Virtual Connection to terminate a Transfer before it has completed, or to terminate a Transfer of unlimited size, or to terminate a persistent memory region. The end device receiving an End operation shall stop sending Control operations (other than End_Ack), and STUs associated with this Transfer. An End kills a Transfer, but shall not affect the Virtual Connection. (See table 4, Com4.)

End may be issued by the Initiator.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id specifies the Responder's Transfer identifier (see 6.2.1).

– I-id specifies the Initiator's Transfer identifier (see 6.2.1).

End_Ack shall be issued by the Responder to confirm that the End operation has been seen and acted on, i.e., acknowledgement that the Transfer has been terminated.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

6.1.2 Write sequence

A Write sequence, as shown in the figure 8 example and detailed in table 5, moves a Transfer, which contains one or more Blocks, from an Initiator to a Responder. A Virtual Connection shall exist before a Write sequence is initiated. Multiple Write sequences can be active at one time in both directions on a single Virtual Connection. (See table 5, W1 through W4.)

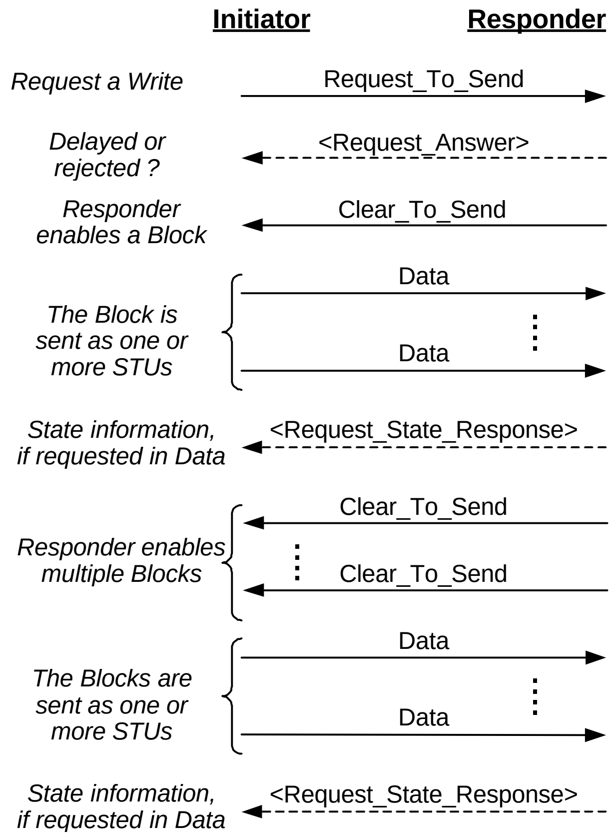


Figure 8 – Write example

Request_To_Send may be issued by the Initiator to request that space be exposed in the Responder for a data Transfer from the Initiator to the Responder.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder. D bits specify the Data Channel to be used.

– CTS_req specifies the number of Blocks that the Initiator would like continuously exposed (see 6.2.10).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id assigns the Initiator's Transfer identifier (see 6.2.1).

– T_len assigns the Transfer length (see 6.2.3).

Request_Answer may be issued by the Responder to reject or pause the Request_To_Send. If rejected, then this is the end of the Write sequence.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. R = 1 specifies that the Request_To_Send has been rejected. R = 0 specifies that the Request_To_Send has been accepted but the subsequent Clear_To_Send operation may be delayed (see 10.1).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

Clear_To_Send operations (one for each Block of the Transfer), shall be issued by the Responder to expose a non-persistent memory region to receive subsequent Data operations. One Block per Clear_To_Send is exposed for a single use.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Mx assigns the Responder's Memory Index for this Block (see 6.2.2).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id assigns the Responder's Transfer identifier (see 6.2.1).

– R-Bufx assigns the Responder's first Buffer Index for this Block (see 6.2.7).

– R-Offset assigns the Responder's initial Offset for this Block (see 6.2.7).

– Blocksize assigns the Block size for this Block (see 6.2.5).

– B_num specifies the Block number for this Block (see 6.2.4).

– F_Offset specifies the initial Offset for the Transfer (see 6.2.7).

Data operations (one for each STU of the Block), shall be issued by the Initiator to send the Block from the data Source (the Initiator), to the data Destination (the Responder). The data Destination shall place the STU data in the memory area pointed to by the Bufx and Offset parameters. The data Destination shall only accept data into pre-allocated memory regions. The data Destination is responsible for ensuring that all of the Blocks of a Transfer are received (see 10.7.8).

– Flags (see 8.2): T = 1 specifies that the data shall be delivered silently. I = 1 specifies that an interrupt shall be generated at the Responder. S = 1 specifies that the Responder shall reply with a Request_State_Response upon successful receipt of this STU. L = 1 marks the last STU of the Block. D bits echo the Data Channel assignment.

– STU_num specifies the number for this STU (see 6.2.6).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– R-Mx echoes the Responder's Memory Index (see 6.2.2).

– R-Bufx assigns the Responder's Buffer Index for this STU (see 6.2.7). If this is the first STU of the Block, then R-Bufx echoes the R-Bufx value in the Clear_To_Send operation.

– R-Offset assigns the Responder's Offset for this STU (see 6.2.7). If this is the first STU of the Block, then R-Offset echoes the R-Offset in the Clear_To_Send operation.

– Sync assigns the Initiator's Sync value (see 6.2.8).

– B_num echoes the Block number for this Block (see 6.2.4).

– Opaque contains the Opaque data for this STU (see 6.2.9).

Request_State_Response shall be issued by the Responder if S = 1 in the previous Data operation.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Slots specifies the number of available Slots in the Responder for this Virtual Connection (see 5.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– B_seq specifies the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly (see 6.2.4).

– Sync echoes the Initiator's Sync value (see 6.2.8).

– B_num indicates if the Block specified in the Data operation with S = 1 was received correctly (see 6.2.4).

An **End** operation may be issued by either end device to abort the Transfer (the Write sequence), or terminate an unlimited size Transfer. (See 6.1.1.4 and table 4, Com4.)

A **Request_State** operation may be issued by the Responder to determine the number of available Slots in the Initiator for this Virtual Connection (to know how many Clear_To_Sends the Responder can issue). (See 6.1.1.1 and table 4, Com1.)

6.1.3 Read sequence

A Read sequence, as shown in the figure 9 example and detailed in table 6, moves a Transfer, which contains one or more Blocks, from a Responder to the Initiator. A Virtual Connection shall exist before a Read sequence is initiated. Multiple Read sequences can be active at one time in both directions on a single Virtual Connection. (See table 6, R1 through R4.)

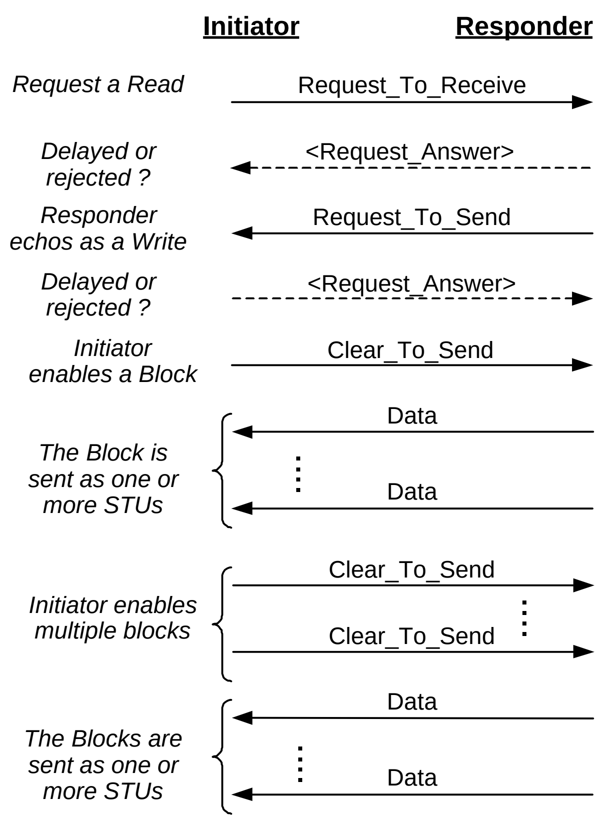


Figure 9 – Read example

Request_To_Receive may be issued by the Initiator to request a data Transfer from the Responder to the Initiator. The Responder shall echo the Request_To_Receive's parameters back in a Request_To_Send operation.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder. D bits specify the Data Channel to be used.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id assigns the Initiator's Transfer identifier (see 6.2.1).

– T_len assigns the Transfer length (see 6.2.3).

Request_Answer may be issued by the Responder to reject or pause the Request_To_Receive. If rejected, then this is the end of the Read sequence.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. R =

1 specifies that the Request_To_Receive has been rejected. R = 0 specifies that the Request_To_Receive has been accepted but the subsequent Request_To_Send operation may be delayed (see 10.1).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

Request_To_Send shall be issued by the Responder to request that space be exposed in the Initiator for a data Transfer from the Responder (the data Source), to the Initiator, (the data Destination).

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. D bits echo the Data Channel to be used.

– CTS_req specifies the number of Blocks that the Responder would like continuously exposed (see 6.2.10).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id assigns the Responder's Transfer identifier (see 6.2.1).

– T_len echoes the Transfer length (see 6.2.3).

Request_Answer may be issued by the Initiator to reject or pause the Request_To_Send. If rejected, then this is the end of the Read sequence.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder. R = 1 specifies that the Request_To_Send has been rejected. R = 0 specifies that the Request_To_Send has been accepted but the subsequent Clear_To_Send operation may be delayed (see 10.1).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

Clear_To_Send operations (one for each Block of the Transfer), shall be issued by the Initiator to expose a non-persistent memory region to receive subsequent Data operations. One Block per Clear_To_Send is exposed for a single use.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder.

– I-Mx assigns the Initiator's Memory Index for this Block (see 6.2.2).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– I-Bufx assigns the Initiator's first Buffer Index for this Block (see 6.2.7).

– I-Offset assigns the Initiator's initial Offset for this Block (see 6.2.7).

– Blocksize assigns the Block size for this Block (see 6.2.5).

– B_num specifies the Block number for this Block (see 6.2.4).

– F_Offset specifies the initial Offset for the Transfer (see 6.2.7).

Data operations (one for each STU of the Block), shall be issued by the Responder to send the Block from the data Source (the Responder), to the data Destination (the Initiator). The data Destination shall place the STU data in the memory area pointed to by the Bufx and Offset parameters. The data Destination shall only accept data into pre-allocated memory regions. The data Destination is responsible for ensuring that all of the Blocks of a Transfer are received (see 10.7.8).

– Flags (see 8.2): T = 1 specifies that the data shall be delivered silently. I = 1 specifies that an interrupt shall be generated at the Initiator. S = 1 specifies that the Initiator shall reply with a Request_State_Response upon successful receipt of this STU. L = 1 marks the last STU of the Block. D bits echo the Data Channel assignment.

– STU_num specifies the number for this STU (see 6.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– I-Mx echoes the Initiator's Memory Index (see 6.2.2).

– I-Bufx assigns the Initiator's Buffer Index for this STU (see 6.2.7). If this is the first STU of the Block, then I-Bufx echoes the I-Bufx value in the Clear_To_Send operation.

– I-Offset assigns the Initiator's Offset for this STU (see 6.2.7). If this is the first STU of the Block, then I-Offset echoes the I-Offset in the Clear_To_Send operation.

– Sync assigns the Responder's Sync value (see 6.2.8).

– B_num echoes the Block number for this Block (see 6.2.4).

– Opaque contains the Opaque data for this STU (see 6.2.9).

An **End** operation may be issued by either end device to abort the Transfer (the Read sequence), or terminate an unlimited size Transfer. (See 6.1.1.4 and table 4, Com4.)

A **Request_State** operation may be issued by the Initiator to determine the number of available Slots in the Responder for this Virtual Connection (to know how many Clear_To_Sends the Initiator can issue). (See 6.1.1.1 and table 4, Com1.)

6.1.4 Put, Get, and FetchOP sequences

The Put, Get, and FetchOP sequences, as shown in the figure 10 example and detailed in table 7, shall be preceded by a sequence that allocates a persistent memory region in the Responder. A Virtual Connection shall exist before a persistent memory region allocation sequence is initiated.

Once allocated, the persistent memory region is available for multiple Put, Get, and FetchOP sequences from the Initiator. By assigning unique values to the G-id and F-id parameters, multiple Get and FetchOP operations may be outstanding to the same persistent memory region. The Put, Get, or FetchOP Blocks shall be contained within a persistent memory region. A Virtual Connection can have multiple persistent memory regions allocated.

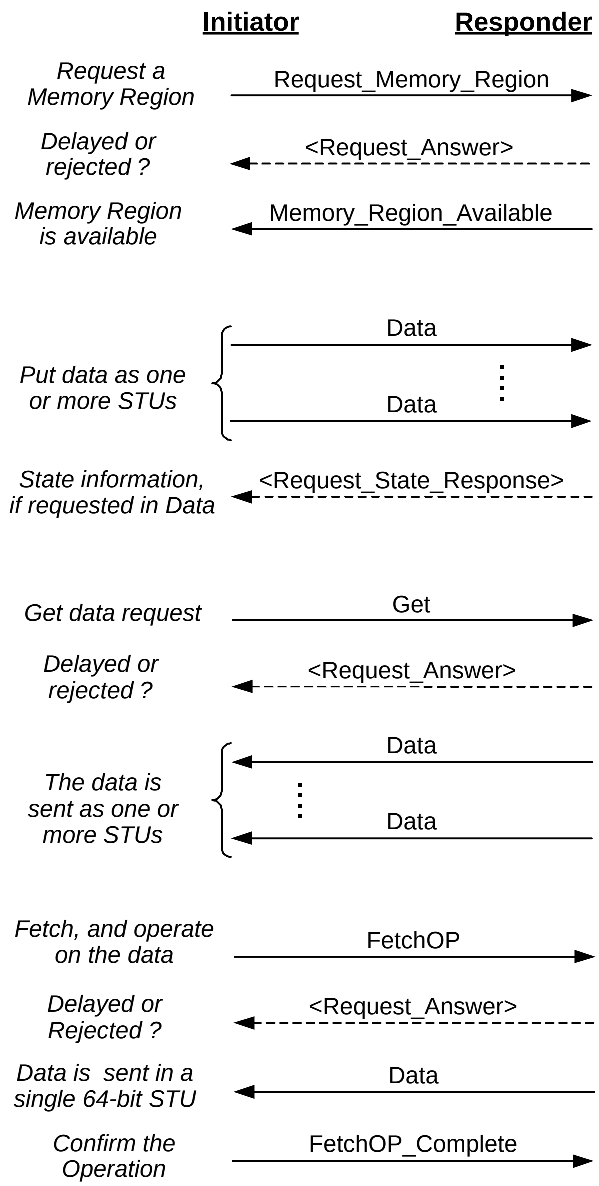


Figure 10 – Put, Get, and FetchOP examples

6.1.4.1 Allocate a persistent memory region

Request_Memory_Region may be issued by the Initiator to request that the Responder allocate a persistent memory region. (See table 7, PG1 and PG2.)

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Responder. D bits specify the Data Channel to be used.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id assigns the Initiator's Transfer identifier (see 6.2.1).

– T_len specifies the persistent memory region's length (see 6.2.3).

Request_Answer may be issued by the Responder to reject or pause the Request_Memory_Region. If rejected, then this is the end of the Put, Get, and FetchOP sequences for this persistent memory region.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. R = 1 specifies that the Request_Memory_Region has been rejected or the Responder does not support persistent memory operations. R = 0 specifies that the Request_Memory_Region has been accepted but the subsequent Memory_Region_Available operation may be delayed (see 10.1).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

Memory_Region_Available shall be sent by the Responder upon receipt of an acceptable Request_Memory_Region operation. (Unacceptable requests result in Request_Answer with Reject = 1.) Once established, the persistent memory region (see 6.2.11), shall remain available (for Put, Get, and FetchOP operations), until terminated by an End or Disconnect sequence (which may come from either end of the Virtual Connection). The Put, Get, and FetchOP operations to the persistent memory region are not flow controlled other than by the Slot accounting rules (see 5.2.6).

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Mx assigns the Responders Memory Index for this persistent memory region (see 6.2.2).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id assigns the Responder's Transfer identifier (see 6.2.1).

– R-Bufx assigns the Responder's first Buffer Index for this persistent memory region (see 6.2.7).

– R-Offset assigns the Responder's initial Offset for this persistent memory region (see 6.2.7).

6.1.4.2 Put sequences

A Put sequence moves a single Block from the Initiator (the data Source), to the Responder (the data Destination). The Block size may be the same, or smaller than, the size of the persistent memory region. If smaller, different data Destination Bufx and Offset values than those specified in the Memory_Region_Available operation, may be used. The Responder shall only accept data into the pre-allocated persistent memory region. (See table 7, PG3 and PG4.)

Data operations (one for each STU of the Block), shall be issued by the Initiator to send the Block being "Put", from the data Source (the Initiator), to the data Destination (the Responder).

– Flags (see 8.2): T = 1 specifies that the data shall be delivered silently. I = 1 specifies that an interrupt shall be generated at the Responder. S = 1 specifies that the Responder shall reply with a Request_State_Response upon successful receipt of this STU. L = 1 marks the last STU of the Block. D bits echo the Data Channel assignment.

– STU_num specifies the number for this STU (see 6.2.6).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– R-Mx echoes the Responder's Memory Index (see 6.2.2).

– R-Bufx assigns the Responder's Buffer Index for this STU (see 6.2.7).

– R-Offset assigns the Responder's Offset for this STU (see 6.2.7).

– Sync assigns the Initiator's Sync value (see 6.2.8).

– B_num specifies the Block number for this Block (see 6.2.4).

– Opaque contains the Opaque data for this STU (see 6.2.9).

Request_State_Response shall be issued by the Responder if S = 1 in the previous Data operation.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator.

– R-Slots specifies the number of available Slots in the Responder for this Virtual Connection (see 5.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– I-id echoes the Initiator's Transfer identifier (see 6.2.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– B_seq specifies the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly (see 6.2.4).

– Sync echoes the Initiator's Sync value (see 6.2.8).

– B_num indicates if the Block, specified in the Data operation with S = 1, was received correctly (see 6.2.4).

An **End** operation may be issued by either end device to abort the Put sequence and terminate the persistent memory region. (See 6.1.1.4 and table 4, Com4.)

6.1.4.3 Get sequences

A Get sequence moves a single Block from the Responder (the data Source), to the Initiator (the data Destination). The Block size may be the same, or smaller than, the size of the persistent memory region. If smaller, different data Source Bufx and Offset values than those specified in the Memory_Region_Available operation may be used. The Responder shall only send data from the pre-allocated persistent memory region. (See table 7, PG5.)

Get may be issued by the Initiator to specify both the Initiator's and Responder's Memory Index, Buffer Index, and Offset values.

– Flags (see 8.2): F = b'000'. I = 1 specifies that an interrupt shall be generated at the Responder. D bits echo the Data Channel assignment.

– I-Mx assigns the Initiator's Memory Index for this Block (see 6.2.2).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– G-id assigns the Initiator's Get identifier (see 6.2.1).

– R-Bufx assigns the Responder's first Buffer Index for this Block (see 6.2.7).

– R-Offset assigns the Responder's initial Offset for this Block (see 6.2.7).

– R-Mx echoes the Responder's Memory Index (see 6.2.2).

– I-Bufx assigns the Initiator's first Buffer Index for this Block (see 6.2.7).

– T_len assigns the Block length (see 6.2.3). Note that a 16-bit length parameter is used, rather than the 64-bit length parameter of most other operations.

– I-Offset assigns the Initiator's initial Offset for this Block (see 6.2.7).

Request_Answer may be issued by the Responder to reject or pause the Get operation. If rejected, then this is the end of the Get sequence.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. R = 1 specifies that the Get operation has been rejected. R = 0 specifies that the Get has been accepted but the subsequent Data operation may be delayed (see 10.1).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– G-id echoes the Initiator's Get identifier (see 6.2.1).

Data operations (one for each STU of the Block), shall be issued by the Responder to send the Block from the data Source (the Responder), to the data Destination (the Initiator). The data Destination shall place the STU data in the pre-allocated memory area pointed to by the I-Bufx and I-Offset parameters. The data Source shall only send data from the pre-allocated persistent memory region.

– Flags (see 8.2): T = 1 specifies that the data shall be delivered silently. I = 1 specifies that an interrupt shall be generated at the Initiator. L = 1 marks the last STU of the Block. D bits echo the Data Channel assignment.

– STU_num specifies the number for this STU (see 6.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– G-id echoes the Initiator's Get identifier (see 6.2.1).

– I-Mx echoes the Initiator's Memory Index (see 6.2.2).

– I-Bufx assigns the Initiator's Buffer Index for this STU (see 6.2.7). If this is the first STU of the Block, then I-Bufx echoes the I-Bufx value in the Get operation.

– I-Offset assigns the Initiator's Offset for this STU (see 6.2.7). If this is the first STU of the Block, then I-Offset echoes the I-Offset in the Get operation.

– Sync assigns the Responder's Sync value (see 6.2.8).

– Opaque contains the Opaque data for this STU (see 6.2.9).

An **End** operation may be issued by either end device to abort the Get sequence and terminate the persistent memory region. (See 6.1.1.4 and table 4, Com4.)

6.1.4.4 FetchOP sequences

A FetchOP sequence fetches data from, and then operates on, a 64-bit aligned, 64-bit data Block in an established persistent memory region in the Responder. The fetch and operation of a FetchOP shall be atomic. The data received by the Initiator shall be the value before the operation is performed. A FetchOP may be retried if the associated Data operation is not returned within the Timeout period (see 10.1). The Responder shall provide the ability to return the same value, upon receipt of subsequent FetchOPs with the same F-id, until the operation is acknowledged with a FetchOP_Complete. Upon receipt of FetchOP_Complete, the Responder shall release its ability to retry the original FetchOP. (See table 7, PG6.)

FetchOP may be issued by the Initiator to specify both the Initiator's and Responder's Buffer Index and Offset values, and the function to be performed on the 64-bit Block.

– Flags (see 8.2): F = function to be performed. I = 1 specifies that an interrupt shall be generated at the Responder. D bits echo the Data Channel assignment.

– I-Mx assigns the Initiator's Memory Index for this Block (see 6.2.2).

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– F-id assigns the Initiator's FetchOP identifier (see 6.2.1).

– R-Bufx assigns the Responder's Buffer Index for this Block (see 6.2.7).

– R-Offset assigns the Responder's Offset, evenly divisible by 8, for this Block (see 6.2.7).

– R-Mx echoes the Responder's Memory Index (see 6.2.2).

– I-Bufx assigns the Initiator's Buffer Index for this Block (see 6.2.7).

– Op_len field = x'0008', i.e., Block length = 8 bytes (see 6.2.3). Note that a 16-bit length parameter is used, rather than the 64-bit length parameter of most other operations.

– I-Offset assigns the Initiator's Offset for this Block (see 6.2.7).

Request_Answer may be issued by the Responder to reject or pause the FetchOP. If rejected, then this is the end of the FetchOP sequence. FetchOP operations shall be rejected by the Responder with a Request_Answer operation with Reject = 1 if: T_len ≠ x'0008', the Responder does not implement atomic FetchOP operations, or a bad parameter, e.g., Bufx or Offset, is supplied in the FetchOP operation.

– Flags (see 8.2): I = 1 specifies that an interrupt shall be generated at the Initiator. R = 1 specifies that the FetchOP has been rejected. R = 0 specifies that the FetchOP has been accepted but the subsequent Data operation may be delayed (see 10.1).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– F-id echoes the Initiator's FetchOP identifier (see 6.2.1).

A **Data** operation shall be issued by the Responder to send the 64-bit Block in one STU from the data Source (the Responder), to the data Destination (the Initiator). The data Destination shall place the STU data in the pre-allocated memory area pointed to by the I-Bufx and I-Offset parameters. The data Source shall only send data from the pre-allocated persistent memory region.

– Flags (see 8.2): T = 1 specifies that the data shall be delivered silently. I = 1 specifies that an interrupt shall be generated at the Initiator. L shall = 1 to specify that this is the last STU of the Block. D bits echo the Data Channel assignment.

– Param field = x'0000', i.e., a FetchOP contains only one STU, so STU_num = x'0000' (see 6.2.6).

– I-Port, R-Port, and I-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– F-id echoes the Initiator's FetchOP identifier (see 6.2.1).

– I-Mx echoes the Initiator's Memory Index (see 6.2.2).

– I-Bufx echoes the Initiator's Buffer Index for this STU (see 6.2.7).

– I-Offset echoes the Initiator's Offset for this STU (see 6.2.7).

– Sync assigns the Responder's Sync value (see 6.2.8).

– Opaque contains the Opaque data for this STU (see 6.2.9).

FetchOP_Complete shall be issued by the Initiator to indicate receipt of the 64-bit data Block. Upon receipt, the Responder shall release its ability to retry the original FetchOP.

– Flags (see 8.2): F = b'111'. I = 1 specifies that an interrupt shall be generated at the Responder.

– R-Port, I-Port, and R-Key echo the Port and Key values assigned during Virtual Connection setup (see 5.1.1).

– R-id echoes the Responder's Transfer identifier (see 6.2.1).

– F-id echoes the Initiator's FetchOP identifier (see 6.2.1).

– Sync echoes the Responder's Sync value (see 6.2.8).

An **End** operation may be issued by either end device to abort the FetchOP sequence and terminate the persistent memory region. (See 6.1.1.4 and table 4, Com4.)

6.2 Data movement parameters

6.2.1 Transfer identifiers (F-id, G-id, I-id, and R-id)

Like the Ports and Keys, each end device shall also select its own 16-bit Transfer identifier for a data movement on the Virtual Connection. x'FFFF' is a reserved value. The Transfer identifiers are:

– F-id = FetchOP Transfer identifier. F-id shall be assigned by the Initiator and passed to the Responder in FetchOP operations. The Responder shall return the F-id in all subsequent operations concerning this FetchOP sequence.

– G-id = Get Transfer identifier. G-id shall be assigned by the Initiator and passed to the Responder in Get operations. The Responder shall return the G-id in all subsequent operations concerning this Get sequence.

– I-id = Initiator's Transfer identifier. I-id shall be assigned by the Initiator and passed to the Responder in Request_To_Send, Request_To_Receive, and Request_Memory_Region operations. The Responder shall return the I-id in all subsequent operations concerning this Transfer or persistent memory region.

– R-id = Responder's Transfer identifier. R-id shall be assigned by the Responder and passed to the Initiator in Clear_To_Send, Request_To_Send, and Memory_Region_Available operations. The Initiator shall return the R-id in all subsequent operations concerning this Transfer or persistent memory region.

In a Request_State operation, D_id field = x'FFFF' means that the Responder shall not look for a current Transfer and shall only return the current number of empty Slots for this Virtual Connection. If a Request_State operation contains an R-id value that the Responder does not recognize, perhaps due to prior completion or termination of the Transfer, then the Responder shall return x'FFFF' in the S_id field of the Request_State_Response operation.

6.2.2 Memory Index (Mx)

Like the Ports, Keys, and Transfer identifiers, each end device shall also select its own 16-bit Memory Index (Mx). The Mx parameter provides a mechanism to identify an area of memory associated with a Block or persistent memory region. A data Destination shall assign locally significant Mx values in each Clear_To_Send, Memory_Region_Available, Get, and FetchOP operation that the data Destination issues. Each assignment may use a different Mx value. The Data operations on this memory region shall use the assigned Mx value. The Mx values are opaque at the data Source, i.e., have no meaning.

6.2.3 Transfer length (T_len)

The 64-bit Transfer length parameter (T_len) in Read and Write sequences specifies the total number of data payload bytes in the Transfer. T_len does not include the Schedule Header or any LLP headers. T_len = all zeros shall indicate an unlimited size Transfer. An unlimited size Transfer may be terminated by an End sequence (see 6.1.1.4).

The 64-bit T_len parameter in a Request_Memory_Region operation specifies the size of the persistent memory region. The size of the persistent memory region is independent of the maximum Block size specified in 5.2.5.

A 16-bit T_len parameter specifies the Block size in Get and FetchOP sequences, exclusive of the Schedule Header or any LLP headers. The Block size in Get and FetchOP sequences shall be ≤ the size of the persistent memory region, and shall conform to the maximum Block size specification in 5.2.5.

6.2.4 Blocks, B_num and B_seq

Scheduled Transfer flow control, striping, acknowledgments, and resource allocation are all done on a Block basis. Block numbers (B_num) for Read sequences, Write sequences, or a set of Puts to a persistent memory region, shall be numbered starting at zero and shall increment by one for each following Block. B_num shall wrap from x'FFFFFFFE' to x'00000000'.

B_num = x'FFFFFFFF' is reserved for use as a flag by Request_State and Request_State_Response operations. In a Request_State operation, B_num = x'FFFFFFFF' indicates that the requestor is not asking if a particular Block has been received correctly. In a Request_State_Response operation of a Request Block state, B_num = x'FFFFFFFF' indicates that the Block (identified by the B_num parameter in the Request_State operation), has not been correctly received by the data Destination.

Blocks shall be enabled for transmission in sequential order unless both end devices indicated Out_of_Order capability during the Virtual Connection setup. Note that Out_of_Order capability is necessary for retransmission to correct flawed Blocks.

The Blocks associated with Read and Write sequences are not persistent. This means that once exposed with a Clear_To_Send operation, a Block can be used only once and may be allocated for other uses after that Block is complete.

Request_State_Response operations indicate (in the B_seq parameter), the highest numbered Block of the specified Transfer received correctly where all lower numbered Blocks are also received correctly. For example, if Blocks 0 through 9 have been received and only Block 6 had an error, then the Request_State_Response operation would have B_num = 5. B_seq = x'FFFFFFFF' indicates that no Blocks have been received by the data Destination for this Transfer.

Request_State_Response operations can be requested by setting the Send_State flag bit in Data operations or by sending Request_State operations. In addition, Request_State operations can ask if a particular Block (B_num) was received correctly for the specified Transfer. Use of these mechanisms allows the data Source to verify correct reception and to identify flawed Blocks for potential retransmission.

6.2.5 Blocksize

The maximum number of bytes in a Block in a Read or Write Transfer (and not applicable for persistent memory regions), is established when the Transfer is initiated. This number is the Blocksize parameter and is expressed as a power of two, i.e., $2^{\text{Blocksize}}$ where $8 \le$ Blocksize ≤ 63. All of the Blocks of a Transfer shall be the same size, except for the first and/or last Block of a Transfer which can be smaller.

The size of the first Block shall be:

Blocksize - (Offset mod Blocksize)

unless the Transfer length is less than this value in which case the first Block contains the entire Transfer. When Blocksize $\ge$ Bufsize, this rule forces the first Block to end on a buffer boundary and makes all subsequent Offsets zero. When Bufsize > Blocksize, this rule forces the first Block to end on one of the hypothetical 2^k Block boundaries within the buffer that would exist if the Offset were zero, thus making all subsequent Offsets some multiple of the Blocksize. The last Block will be whatever completes the Transfer.

6.2.6 STUs and STU_num

For performance reasons the STUs shall be transmitted in order. STU numbers (STU_num) for a Block shall start with zero and increment by one for each following STU. The last STU of a Block shall be marked with Last = 1. No STU shall extend past a data Destination's buffer boundary, Block boundary, or Transfer boundary. Out of order delivery may cause errors or reduce performance (see 10.7.4). There is no requirement that STU sizes be consistent throughout a Block or Transfer, but STU sizes shall be no larger than specified in 5.2.4.

6.2.7 Bufx and Offset

Bufx contains a Buffer Index. If more than one Buffer Index is required for a Block, i.e., buffer size (Bufsize) is less than Blocksize, then the Bufx parameter in a Clear_To_Send, Get, or FetchOP operation shall specify the initial Bufx, and any additional Bufx values shall be sequential.

Offset may be used to start at other than the first byte of a buffer. For the first STU of a Block, the Offset shall be the same as received in the Clear_To_Send, Get, or FetchOP for the Block. Subsequent STUs of the Block shall adjust the Bufx and Offset based on the data Destination's buffer size and the STU size used by the data Source.

The Offset associated with the first block of a Transfer (F_Offset) is included in all Clear_To_Send operations. This allows the data Destination to compute the starting address for any Block without having received the Clear_To_Send for the first Block. Clear_To_Send operations can occur out of order, e.g., as the result of striping (see annex B).

Best performance will usually be achieved when an Offset of zero is specified. Use of non-zero Offset may degrade performance, depending upon underlying hardware transfer mechanisms.

6.2.8 Sync

The 32-bit Sync parameter shall be used to associate a Request_State_Response operation with one of many Data or Request_State solicitations (see 5.2.6 concerning updating a local end device's image of the number of available Slots at the remote end device for this Virtual Connection). Sync shall also be used to associate a FetchOP_Complete operation with a Data operation.

The Sync parameter provides a mechanism for a data Source to identify and track subsequent operations within a sequence. These Sync values have no meaning at the data Destination. A data Source shall assign a locally significant Sync value in each Data operation that the data Source issues. Each assignment may use a different Sync value. The associated Request_State_Response and FetchOP_Complete operations shall echo the Sync value.

A local end device (regardless of whether it is the data Source or data Destination), shall assign a locally significant Sync value in each Request_State operation that the local end device issues. Each assignment may use a different Sync value. The associated Request_State_Response shall echo the Sync value.

6.2.9 Opaque data

Opaque data is six bytes of ULP peer-to-peer information carried in a Data operation’s Schedule Header Op_len and Offset_2 fields. The Opaque data shall be delivered to the recipient’s ULP when Silent = 0 or Send_State = 1 (see 8.2). The Opaque data shall be passed, unmodified by any intermediate device, from the Source to the Destination. Note that the Opaque data uses Slot resources while the data payload uses Bufx resources. The Opaque data shall not be counted in the length, tiling, or Bufx calculations.

6.2.10 Blocks enabled (CTS_req)

In Read and Write sequences, the data Source specifies in the Request_To_Send operation the number of Blocks that the data Source would like to see continuously exposed for maximum performance. Since each Block is exposed for a single use with a Clear_To_Send operation, this parameter is named CTS_req. CTS_req = x'0000' means don't care. The CTS_req is "advice" to improve performance; it is not mandatory that the other end comply by issuing that number of Clear_To_Send operations. Specifying the number of simultaneously enabled Blocks is useful for pipelining and striping where multiple Blocks can be en-route simultaneously (see annex B).

6.2.11 Persistent memory

The memory region associated with a Request_Memory_Region operation is persistent. That means that once allocated, the memory region remains available for multiple Put, Get, and FetchOP sequences until released by an End or Port teardown sequence.

For Get and FetchOP sequences, the Responder's transmit and receive buffer sizes shall the same size. Note that the Initiator must know the Responder's transmit buffer size to correctly calculate the R-Bufx and R-Offset values, and only an end device's receive buffer size is exchanged during Virtual Connection setup (see 5.1.1 and 5.2.3).

6.3 Packing examples

Figure 11 shows three possibilities for packing the same Transfer into a data Destination's buffers. All three examples show a group of seven of the data Destination's buffers on the top line. Each buffer is pointed to by a Bufx, and the data in the first buffer starts at an Offset. The Transfer is the shaded bar, with transmission going from left to right. The Block boundaries are shown above the shaded bar, and the resulting STU boundaries are shown below the shaded bar.

Example (a), at the top, shows the case where the buffers and Blocks are the same size. Notice that the first Block is smaller than the other Blocks by the Offset. Offset = zero for the other Blocks. The last Block of the Transfer is also smaller, i.e., the Transfer did not end on a Block boundary. While the STU boundaries lined up nicely, the issuer could have used multiple STUs, but the STUs cannot be larger than Max_STU.

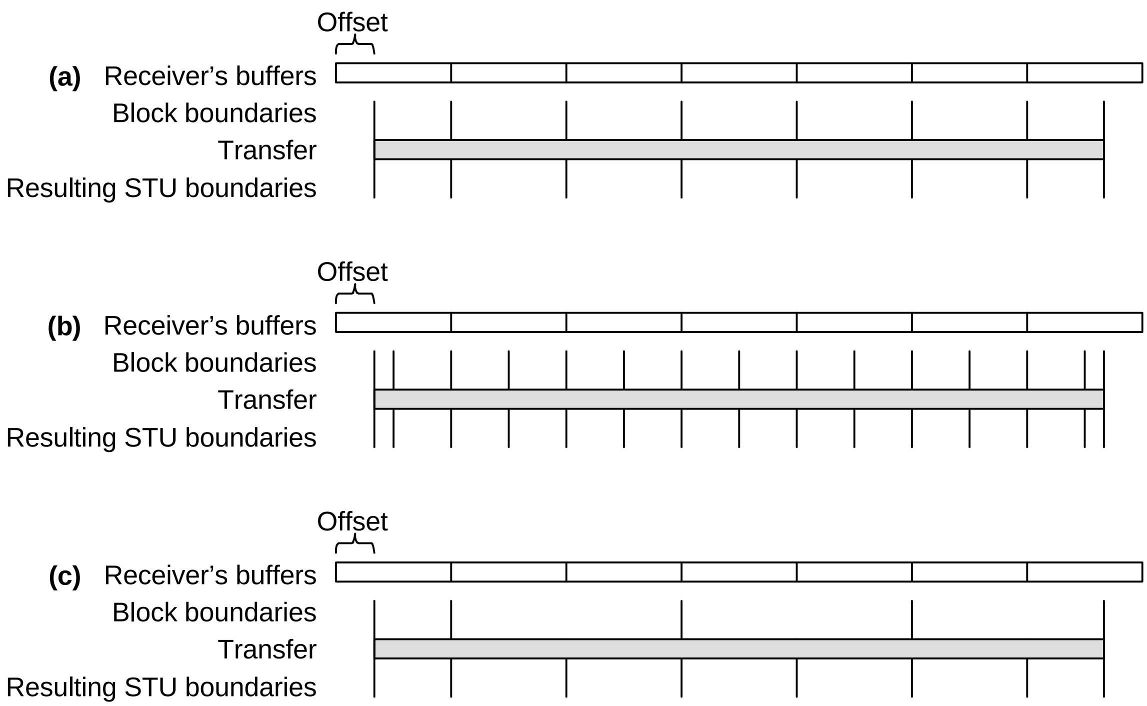


Figure 11 – Data packing examples

Example (b) shows multiple Blocks per receiver buffer. The Blocks that do not start on a buffer boundary would use the Offset parameter to position the data.

Example (c) shows the Block size covering two of the receiver's buffers.

In summary, STUs cannot cross Block, buffer, or Transfer boundaries. Relationships include:

STU size $\le 2^{Max\_STU}$

Max_STU $\le$ Blocksize

Max_STU $\le$ Bufsize

Note that the Blocksize can be larger, smaller, or the same as Bufsize.

7 Operations management

7.1 Flow control

Data flow control in Read and Write sequences is achieved with Clear_To_Send operations; each Clear_To_Send received gives the data Source permission to send one Block one time. There is no equivalent flow control for Put, Get, and FetchOP operations.

Operation flow control is achieved by an operation's issuer not overrunning the Slots value (see 5.2.6).

7.2 Status operations

Request_State and Request_State_Response operations are used to request and supply status information about the state of the remote end device. They can be used to see which Blocks have been received correctly for a specific Transfer, and the number of empty Slots available for this Virtual Connection. The Sync parameter (see 5.2.6) is used to provide a common reference point for the local and remote end devices, i.e., to match Request_State and Request_State_Response operations.

7.3 Rejected operations

If the receiving end device is unable to execute an operation, then the receiving device shall set the Reject flag bit = 1 in the response. Table 1 shows the response when an operation is rejected. The recovery actions taken when an operation is rejected are beyond the scope of this standard.

Table 1 – Response to a rejected operation

Rejected operation	**Response** (w/ Reject=1)
Request_Connection	Connection_Answer
Request_To_Send	Request_Answer
Request_To_Receive	Request_Answer
Request_Memory_Region	Request_Answer
Get	Request_Answer
FetchOP	Request_Answer

7.4 Interrupts

An Interrupt causes a signal to be delivered to the receiving end device ULP. An Interrupt can be requested with any operation by setting Interrupt = 1.

8 Schedule Header

The Schedule Header is shown in figure 12 as a group of 32-bit words. The Schedule Header fields are named for the most common parameter for which the field is used. Many of the fields have different uses depending on the operation type, and some operations do not use one or more of the fields at all. The usage for each field is specified in tables 3-7.

If an operation does not use a particular Schedule Header field, then that field shall be transmitted as zeros. If a parameter does not completely fill a field then the parameter shall be right justified with leading zeros used to pad out the field.

			Bytes
Op	Flags	Param	00-03
D_Port		S_Port	04-07
D_Key			08-11
D_id		S_id	12-15
Bufx			16-19
Offset			20-23
Sync			24-27
B_num			28-31
Cksum		Op_len	32-35
Offset_2			36-39

Figure 12 – Schedule Header contents

8.1 Op codes

The operations, and their 5-bit Op code are listed in table 2. Unspecified Op values are reserved.

Table 2 – Op codes and operations

Op	Operation
x'01'	Request_Connection
x'02'	Connection_Answer
x'03'	Request_Disconnect
x'04'	Disconnect_Answer
x'05'	Disconnect_Complete
x'13'	Request_Memory_Region
x'14'	Memory_Region_Available
x'15'	Get, FetchOP, FetchOP_Complete
x'16'	Request_To_Send
x'17'	Request_Answer
x'18'	Request_To_Receive
x'1A'	Clear_To_Send
x'1B'	Data
x'1C'	Request_State
x'1D	Request_State_Response
x'1E'	End
x'1F'	End_Ack

8.2 Flags

Figure 13 shows the flags, and their relative position within the Flags field. The flag functions are detailed below for the case where the bit = 1. The Flags field column in tables 3-7 specify the flags that are valid for each operation.

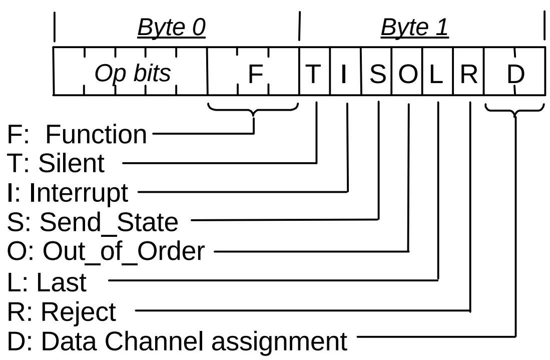


Figure 13 – Flags summary

Function: In FetchOP operations, the Function flags shall have the following meanings. The unspecified values are reserved.

b'000xxxxxxxx' = NOP (i.e., Get)
b'001xxxxxxxx' = Fetch and Increment
b'010xxxxxxxx' = Fetch and Decrement
b'011xxxxxxxx' = Fetch and Clear
b'111xxxxxxxx' = FetchOP Complete

In Request_Connection and Connection_ Answer operations, the Function flags specify the issuing end device's attributes:

b'x00xxxxxxxx' = Does not support persistent memory
b'x01xxxxxxxx' = Supports persistent memory but not FetchOP operations
b'x11xxxxxxxx' = Supports persistent memory and FetchOP operations
b'1xxxxxxxxxx' = Little endian ULP architecture

Silent (b'xxx1xxxxxxx') = Requests silent delivery of a Data operation (see 5.2.6). When Silent = 1 the data transfer to the data Destination Bufx is carried out normally, but the Schedule Header shall not be delivered to any ULP. This provides the basis for remote memory write semantics where the intent is to modify the contents of a remote memory without executing software in the remote host computer. This also provides a means for reducing overhead by suppressing all but the final Schedule Header delivery to the ULP during a lengthy Scheduled Transfer. Silent is overridden when Send_State = 1.

Interrupt (b'xxxx1xxxxxx') = Requests that a signal or interrupt be generated and delivered to the appropriate ULP. The Interrupt flag is independent of the Silent flag, i.e., Interrupt = 1 calls for a signal whether or not Silent = 1. (See 7.5.)

NOTE 1 – The Silent and Interrupt flags together provide for three delivery modes for Data operations: silent, polled, or interrupt-driven. If Silent = 1, the data payloads are delivered silently. If Silent = 0, then the ULP is informed by the same means used for all other Schedule Headers and the Schedule Header for this operation is delivered to the ULP. This mode is suitable for polled interfaces. If Interrupt = 1, then a signal is delivered.

Send_State (b'xxxxx1xxxxx') = Requests that the receiving ULP respond with a Request_State_Response upon successful receipt of this STU. Send_State is valid in all Data operations. Send_State = 1 overrides the Silent flag actions, and always consumes a Slot.

Out_of_Order (b'xxxxxx1xxxx') = The Source is able to send and receive Blocks in any order.

Last (b'xxxxxxx1xxx') = Marks the last STU of a Block.

Reject (b'xxxxxxxx1xx') = The Request_ Connection, Request_To_Send, Request_To_ Receive, Get, Request_Memory_Region, or FetchOP operation has been rejected.

Data Channel assignment: The Data Channel to be used to carry Data operations. The Data Channel value is assigned in a Request_To_Send, Request_To_Receive, or Request_Memory_Region operation, and is the Data Channel to be used for Data operations associated with this Transfer. The Data Channel Assignment is not checked at the data Destination.

b'xxxxxxxxx01' = Data Channel 1
b'xxxxxxxxx10' = Data Channel 2
b'xxxxxxxxx11' = Data Channel 3

The maximum STU size sent on Data Channels 1 and 2 shall be 2^{17} bytes (i.e., 128

Kbytes). The maximum STU size sent on Data Channel 3 shall be 2^{31} bytes (i.e., 2 gigabytes).

NOTE 2 – Data Channel assignment value b'00' is reserved.

8.3 Checksum (optional)

The 16-bit end-to-end checksum (Cksum) shall be transmitted in every operation, and optionally checked (see 10.4) at each Destination. End device that support checksums shall compute Cksum as the one's complement of the one's complement sum of all of the bytes (taken two bytes, or 16 bits at a time), in the operation, including the ST Header with zeros in the Cksum field. If the computed Cksum = x'0000', then Cksum = x'FFFF' shall be transmitted.

If a Source does not support checksums, then it shall transmit Cksum = x'0000'.

NOTE – The checksum algorithm is basically the same as used with the Internet Protocol (i.e., IP, RFC 791), Transmission Control Protocol (i.e., TCP, RFC 793), and User Datagram Protocol (i.e., UDP, RFC 768). Transmitting Cksum = x'FFFF' in place of a computed Cksum = x'0000', and transmitting Cksum = x'0000' to indicate non-support of checksums, is the same as in UDP. Also, note RFC 1936, *Implementing the Internet Checksum in Hardware*.

Open Issue – The checksum algorithm is preliminary and may change as the result of further investigation, e.g., comparisons to the ISO and Fletcher checksums.

9 Operations details

Tables 3-7 define the parameters that shall be carried in each field of the Schedule Header for each operation. Within the operations, the following prefixes are used:

I- = associated with the Initiator

R- = associated with the Responder

Other rules associated with the tables include:

– Operations contained within <…> are conditional, and may not occur.

– The entries under the Flags parameter are abbreviations for the individual flag bits as shown in figure 13.

– Multiword parameters and field names are joined with an underscore, e.g., D_Port.

– Values in bold italics are assigned by the specific operation and may be used by later operations.

– A * marks a field carrying an unused value; that field shall be transmitted as zeros and shall not be checked at the receiver.

– If a parameter does not completely fill a field then the parameter shall be right justified with leading zeros used to pad out the field.

Table 3 – Connection management sequences

Operation	Issued by	Op	Flags	Param	D_Port	S_Port	D_Key
(C1) A Virtual Connection is set up between the Initiator and Responder end devices							
Request_Connection	Initiator	x'01'	***FIO***	***I-Slots***	***R-Port***	***I-Port***	*
Connection_Answer	Responder	x'02'	***FIOR***	***R-Slots***	I-Port	***R-Port***	I-Key
(C2) A Virtual Connection teardown sequence can be initiated from either end							
Request_Disconnect	Initiator	x'03'	***I***	*	R-Port	I-Port	R-Key
Disconnect_Answer	Responder	x'04'	***I***	*	I-Port	R-Port	I-Key
Disconnect_Complete	Initiator	x'05'	***I***	*	R-Port	I-Port	R-Key
NOTE - The Initiator in C1 and C2 can be either party of the Virtual Connection.							

Table 4 – Common control sequences

Operation	Issued by	Op	Flags	Param	D_Port	S_Port	D_Key
(Com1) Request Slot state: free Slots (a Transfer does not need to be in progress)							
Request_State	Initiator	x'1C'	***I***	*	R-Port	I-Port	R-Key
Request_State_Response	Responder	x'1D'	***I***	***R-Slots***	I-Port	R-Port	I-Key
(Com2) Request Transfer state: free Slots, highest Block received OK							
Request_State	Initiator	x'1C'	***I***	*	R-Port	I-Port	R-Key
Request_State_Response	Responder	x'1D'	***I***	***R-Slots***	I-Port	R-Port	I-Key
(Com3) Request Block state: free Slots, highest Block received OK, specific Block OK?							
Request_State	Initiator	x'1C'	***I***	*	R-Port	I-Port	R-Key
Request_State_Response	Responder	x'1D'	***I***	***R-Slots***	I-Port	R-Port	I-Key
(Com4) Terminate the Transfer and release resources							
End	Initiator	x'1E'	***I***	*	R-Port	I-Port	R-Key
End_Ack	Responder	x'1F'	***I***	*	I-Port	R-Port	I-Key
NOTE - The Initiator in any sequence in this table can be either party of the Virtual Connection.							

Table 3 (cont.) – Connection management sequences

D_id	S_id	Bufx	Offset	Sync	B_num	Cksum	Op_len	Offset_2
(See 5.1.1.)								
*	*	***I-Bufsize***	***I-Key***	***I-Max_STU***	***EtherType***	Cksum	*	***I-Max_Block***
*	*	***R-Bufsize***	***R-Key***	***R-Max_STU***	*	Cksum	*	***R-Max_Block***
(See 5.1.2)								
*	*	*	I-Key	*	*	Cksum	*	*
*	*	*	R-Key	*	*	Cksum	*	*
*	*	*	I-Key	*	*	Cksum	*	*

Table 4 (cont.) – Common control sequences

D_id	S_id	Bufx	Offset	Sync	B_num	Cksum	Op_len	Offset_2
(See 6.1.1.1.)								
x'FFFF'	*	*	*	***Sync***	*	Cksum	*	*
x'FFFF'	*	*	*	Sync	*	Cksum	*	*
(See 6.1.1.2.)								
R-id	I-id	*	*	***Sync***	x'FFFFFFFF'	Cksum	*	*
I-id	R-id	*	***B_seq***	Sync	x'FFFFFFFF'	Cksum	*	*
(See 6.1.1.3.)								
R-id	I-id	*	*	***Sync***	***B_num***	Cksum	*	*
I-id	R-id	*	***B_seq***	Sync	B_num	Cksum	*	*
(See 6.1.1.4.)								
R-id	I-id	*	*	*	*	Cksum	*	*
I-id	R-id	*	*	*	*	Cksum	*	*

Table 5 – Write sequences

Operation	Issued by	Op	Flags	Param	D_Port	S_Port	D_Key
(W1) The Initiator requests to write data (a Transfer) to the Responder							
Request_To_Send	Initiator	x'16'	***ID***	***CTS_req***	R-Port	I-Port	R-Key
<Request_Answer>	Responder	x'17'	***IR***	*	I-Port	R-Port	I-Key
(W2) The Responder exposes a memory region (a Block) to the Initiator							
Clear_To_Send	Responder	x'1A'	***I***	***R-Mx***	I-Port	R-Port	I-Key
(W3) The Initiator sends data (an STU) to the Responder							
Data	Initiator	x'1B'	***TISL***D	***STU_num***	R-Port	I-Port	R-Key
(W4) The Responder sends state information, if so requested in the Data operation							
<Request_State_Response>	Responder	x'1D'	***I***	***R-Slots***	I-Port	R-Port	I-Key
NOTE - The Initiator in this table is the end device that issues the Request_To_Send operation.							

Table 6 – Read sequences

Operation	Issued by	Op	Flags	Param	D_Port	S_Port	D_Key
(R1) The Initiator requests to read data (a Transfer) from the Responder							
Request_To_Receive	Initiator	x'18'	***ID***	*	R-Port	I-Port	R-Key
<Request_Answer>	Responder	x'17'	***IR***	*	I-Port	R-Port	I-Key
(R2) The Responder echoes the Initiator's request as a request to write data							
Request_To_Send	Responder	x'16'	***I***D	***CTS_req***	I-Port	R-Port	I-Key
<Request_Answer>	Initiator	x'17'	***IR***	*	R-Port	I-Port	R-Key
(R3) The Initiator exposes a memory region (a Block) to the Responder							
Clear_To_Send	Initiator	x'1A'	***I***	***I-Mx***	R-Port	I-Port	R-Key
(R4) The Responder sends data (an STU) to the Initiator							
Data	Responder	x'1B'	***TISL***D	***STU_num***	I-Port	R-Port	I-Key
NOTE - The Initiator in this table is the end device that issues the Request_To_Receive operation.							

Table 5 (cont.) – Write sequences

D_id	**S_id**	**Bufx**	**Offset**	**Sync**	**B_num**	**Cksum**	**Op_len**	**Offset_2**
(See 6.1.2.)								
*	***I-id***	*	*	***T_len***		Cksum	*	*
I-id	*	*	*	*	*	Cksum	*	*
(See 6.1.2.)								
I-id	***R-id***	***R-Bufx***	***R-Offset***	***Blocksize***	***B_num***	Cksum	*	***F_Offset***
(See 6.1.2.)								
R-id	R-Mx	***R-Bufx***	***R-Offset***	***Sync***	B_num	Cksum	***Opaque***	
(See 6.1.2.)								
I-id	R-id	*	***B_seq***	Sync	B_num	Cksum	*	*

Table 6 (cont.) – Read sequences

D_id	**S_id**	**Bufx**	**Offset**	**Sync**	**B_num**	**Cksum**	**Op_len**	**Offset_2**
(See 6.1.3.)								
*	***I-id***	*	*	***T_len***		Cksum	*	*
I-id	*	*	*	*	*	Cksum	*	*
(See 6.1.3.)								
I-id	***R-id***	*	*	T_len		Cksum	*	*
R-id	*	*	*	*	*	Cksum	*	*
(See 6.1.3.)								
R-id	I-id	***I-Bufx***	***I-Offset***	***Blocksize***	***B_num***	Cksum	*	***F_Offset***
(See 6.1.3.)								
I-id	I-Mx	***I-Bufx***	***I-Offset***	***Sync***	B_num	Cksum	***Opaque***	

Table 7 – Put, Get, and FetchOP sequences

Operation	**Issued by**	**Op**	**Flags**	**Param**	**D_Port**	**S_Port**	**D_Key**
(PG1) The Initiator requests a persistent memory region on the Responder							
Request_Memory_Region	Initiator	x'13'	***ID***	*	R-Port	I-Port	R-Key
<Request_Answer>	Responder	x'17'	***IR***	*	I-Port	R-Port	I-Key
(PG2) The Responder allocates a persistent memory region (a Block) to the Initiator							
Memory_Region_Available	Responder	x'14'	***I***	***R-Mx***	I-Port	R-Port	I-Key
(PG3) The Initiator Puts data (an STU) in the Responder's persistent memory region							
Data	Initiator	x'1B'	***TISL***D	***STU_num***	R-Port	I-Port	R-Key
(PG4) The Responder sends state information, if so requested in the Data operation							
<Request_State_Response>	Responder	x'1D'	***I***	***R-Slots***	I-Port	R-Port	I-Key
(PG5) The Initiator Gets data from the Responder's persistent memory region							
Get	Initiator	x'15'	***FI***D	***I-Mx***	R-Port	I-Port	R-Key
<Request_Answer>	Responder	x'17'	***IR***	*	I-Port	R-Port	I-Key
Data	Responder	x'1B'	***TIL***D	***STU_num***	I-Port	R-Port	I-Key
(PG6) The Initiator fetches and operates on data in the Responder's persistent memory							
FetchOP	Initiator	x'15'	***FI***D	***I-Mx***	R-Port	I-Port	R-Key
<Request_Answer>	Responder	x'17'	***IR***	*	I-Port	R-Port	I-Key
Data	Responder	x'1B'	***TIL***D	x'0000'	I-Port	R-Port	I-Key
FetchOP_Complete	Initiator	x’15’	***FI***	*	R-Port	I-Port	R-Key
NOTE - The Initiator is the end device that issues the Request_Memory_Region operation.							

Table 7 (cont.) – Put, Get, and FetchOP sequences

D_id	**S_id**	**Bufx**	**Offset**	**Sync**	**B_num**	**Cksum**	**Op_len**	**Offset_2**
(See 6.1.4.1.)								
*	***I-id***	*	*	***T_len***		Cksum	*	*
I-id	*	*	*	*	*	Cksum	*	*
(See 6.1.4.1.)								
I-id	***R-id***	***R-Bufx***	***R-Offset***	*	*	Cksum	*	*
(See 6.1.4.2.)								
R-id	R-Mx	***R-Bufx***	***R-Offset***	***Sync***	***B_num***	Cksum	***Opaque***	
(See 6.1.4.2.)								
I-id	R-id	*	***B_seq***	Sync	B_num	Cksum	*	*
(See 6.1.4.3.)								
R-id	***G-id***	***R-Bufx***	***R-Offset***	R-Mx	***I-Bufx***	Cksum	***T_len***	***I-Offset***
G-id	*	*	*	*	*	Cksum	*	*
G-id	I-Mx	***I-Bufx***	***I-Offset***	***Sync***	*	Cksum	***Opaque***	
(See 6.1.4.3.)								
R-id	***F-id***	***R-Bufx***	***R-Offset***	R-Mx	***I-Bufx***	Cksum	x'0008'	***I-Offset***
F-id	*	*	*	*	*	Cksum	*	*
F-id	I-Mx	I-Bufx	I-Offset	***Sync***	*	Cksum	***Opaque***	
R-id	F-id	*	*	Sync	*	Cksum	*	*

10 Error processing

Table 9 is a summary of the logged errors. The logging shall be on a per-Port basis, and shall be available to the ULP that is using the Port. The nature and size of the logs are system dependent.

Open Issue – The whole error processing clause needs a very detailed review, the editor keeps finding exceptions and errors.

10.1 Operation timeout

Errors other than syntactic errors are manifested as missing operations, occurring when the underlying physical medium discard or damage a transmission. Such errors are detected by Op_timeout, which is system and/or Port dependent. Op_timeout_Occurances shall be logged. Example means for determining the Op_timeout value for a Virtual Connection include:

– a time longer than the measured round-trip time through the software path (use a Request_State / Request_State_Response pair to measure on a per-Port basis); or

– a long fixed time period; or

– a time equal to the maximum queuing delay for a maximum size message (e.g., for a Control operation queued behind a large Transfer).

When reliable data movement operations are required by the ULP, each operation that expects a response shall be guarded with a timeout whose value is Op_timeout. Data transmissions (i.e., Data operations) are an exception to this timeout mechanism and are referred to the ULP for resolution (see 10.7.8). The ULP that issues a Clear_To_Send or a Request_To_Receive is responsible for timing out these operations. The ULP may or may not use Op_timeout to time out Data operations.

Another system and/or Port dependent parameter, Max_Retry, specifies the maximum number of times to retry an operation. If enabled, an operation shall be re-tried up to Max_Retry times if the sending end device does not receive the expected response (see table 8). If Max_Retry is reached without success, then the operation is considered to be aborted and control shall be passed to the ULP. Max_Retry_Occurances shall be logged.

10.2 Operation Pairs

Each Scheduled Transfer operation is defined as part of a two-way handshake or a three-way handshake. Thus, for each command operation there is a corresponding response operation, and for some response operations there is also a corresponding completion operation. Table 8 lists the operation pairs – command and response, or response and completion – that shall be retried if the associated response is not received within an Op_timeout.

Additionally, Request_State_Response is a corresponding pair for Data operations which have Send_State = 1. If the Request_State_Response is not received, then the data Source may send a Request_State to obtain the state information.

Table 8 – Operation pairs guarded by Op_timeout with mandatory retry

Operation	Response(s)
Data (w/ Send_State=1)	Request_State_Response
Disconnect_Answer	Disconnect_Complete
End	End_Ack
FetchOP	Data, or Request_Answer
Get	Data, or Request_Answer
Request_Connection	Connection_Answer
Request_Disconnect	Disconnect_Answer
Request_State	Request_State_Response
Request_To_Send	Request_Answer or Clear_To_Send
Request_To_Receive	Request_Answer or Request_To_Send

10.3 Duplicate operations

Open Issue – Greg Chesson has an action item to draft some text on how you differentiate duplicate operations from legal operations.

10.4 Checksum errors

If an end device that supports checksums computes a Cksum value (see 8.3) different from that received in the operation's ST Header, and the received Cksum ≠ x'0000', then the operation shall be discarded and a Cksum_Error shall be logged.

10.5 Syntax errors

10.5.1 Undefined Opcode

An operation with an undefined Opcode value shall be discarded, an Undefined_Opcode_Error shall be logged, and the Opcode shall be logged in Undefined_Opcode_Value.

10.5.2 Unexpected Opcode

Most of the operations require previous operations to set up state on each device. If a device receives an out of sequence Opcode (e.g., receiving a Connection_Answer without having sent the initiating Request_Connection), the operation shall be discarded, an Unexpected_Opcode_Error shall be logged, and the Opcode shall be logged in Unexpected_Opcode_Value.

10.6 Virtual Connection errors

10.6.1 Invalid Key or Port

All operations, excluding Request_Connection and Disconnect operations, should have a Key (see 5.2.2) value that validates the operation for the Virtual Connection. Operations with an invalid Key shall not be executed, and an Invalid_Key_Error shall be logged.

All operations, excluding Disconnect operations, should have a valid Destination Port value (see 5.2.1). Operations with an invalid Destination Port value shall not be executed, and an Invalid_Port_Error shall be logged.

If a Request_Disconnect is received and the Port and/or Key values are invalid, then a Disconnect_Answer shall be issued. If a Disconnect_Answer is received and the Port and/or Key values are invalid, then a Disconnect_Complete shall be issued. In both cases, the R-Port, I-Port, R-Key and I-Key, values in the received operation shall be used to form the operation issued.

NOTES
1 – Multiple contiguous invalid Key and/or Port values may indicate a problem with the link or a malicious host on the network. The supervising process should be informed.

2 – Since the Disconnect operations have a complete set of parameters for both the Initiator and Responder, legal Disconnect responses can be generated, even if one end device has lost the state information for the Virtual Connection, e.g., due to a power-down.

10.6.2 Slots exceeded

Operations that exceed the number of Slots (see 5.2.6) for the Virtual Connection may not be executed, and a Slots_Exceeded_Error shall be logged.

10.6.3 Unknown EtherType

If a Request_Connection operation contains an unknown EtherType (see 5.2.7), the receiver shall issue a Connection_Answer with Reject = 1 and shall log an Unknown_EtherType_Error.

10.6.4 Illegal Bufsize

If a Request_Connection contains a Bufsize (see 5.2.3) value that is < 8 or > 63, (i.e., Buffer size < 2^8 bytes, or > 2^{63} bytes), then the receiver shall respond with a Connection_Answer with Reject = 1. If a Connection_Answer contains a Bufsize value that is < 8 or > 63, then the receiver shall respond with a Request_Disconnect. In either case, an Illegal_Bufsize_Error shall be logged.

10.6.5 Illegal STU size

The maximum STU sizes (Max_STU) for each end device were determined during the Virtual Connection setup (see 5.1.1 and 5.2.4). If the received STU in a Data operation is greater than the maximum STU size, then the STU shall be discarded and an Illegal_STU_Size_Error shall be logged.

10.7 Scheduled Transfer errors

10.7.1 Invalid Transfer identifier

Many Scheduled Transfer operations use a Transfer identifier (see 6.2.1), in the D_id field for quickly accessing state information for a sequence. An operation with an invalid Transfer identifier shall be discarded and an Invalid_D-id_Error shall be logged.

10.7.2 Invalid Memory Index (Mx)

Data operations echo previously assigned Memory Index (Mx) values (see 6.2.2). If the Mx value in a Data operation does match a valid Mx value, then the Data operation shall be discarded and an Invalid_Mx_Error shall be logged.

10.7.3 Bad Data Channel specification

During Request_To_Send, Request_To_Receive, and Request_Memory_Region operations, the initiating device declares the LLP Data Channel that will carry Data operations for the sequence. Some Data Channels may not be available for Data operations depending on the LLP (e.g., b'00' is not a valid choice on HIPPI-6400 as it indicates VC0 which is reserved for Control operations). If the Data Channel value is in error, then the receiver shall issue an appropriate response with Reject = 1.

10.7.4 Out of Range B_num, Bufx, Offset, or STU_num

If the Block number (see 6.2.4) in a received:

– Clear_To_Send operation is outside the calculated number of Blocks for the Transfer;

– Data or Request_State operation, to other than a persistent memory region, has not been previously exposed by a Clear_To_Send operation;

then the offending operation shall be discarded and an Out_Of_Range_B_num_Error shall be logged.

If a Data, Get, or FetchOP operation contains a Bufx and/or Offset (see 6.2.7) that exceeds the buffer range allocated by the data Destination, then the receiver shall discard the operation and shall log an Out_Of_Range_Bufx_Error.

If a Data, Get, or FetchOP operation contains an Offset (see 6.2.7) larger than the buffer size, the receiver shall discard the operation and shall log an Oversized_Offset_Error.

If a Data operation contains an STU_num (see 6.2.6) that is not one greater than the previous STU for this Block, then the STU is out of order. The receiver may discard the STU and log an Out_Of_Order_STU_Error if it cannot accommodate out of order STU delivery.

10.7.5 Block out of order error

If a Data operation contains a B_num that is not one greater than the previous B_num for this Transfer or persistent memory region, and Out_of_Order (see 8.2) capability was not specified during the Virtual Connection setup (see 5.1.1), then the data Destination shall log an Out_Of_Order_B_num and may terminate the Transfer or persistent memory region with an End sequence.

10.7.6 Illegal Blocksize

If a Clear_To_Send operation contains a Blocksize (see 6.2.5) value that is < 8 or > 63, (i.e., Block size < 2^8 bytes, or > 2^{63} bytes), then the receiver shall discard the operation and shall log an Illegal_Blocksize_Error.

Open Issue – The first "shall" above was previously a "should". The change needs to be checked for correctness.

10.7.7 Undefined Flag

If a received operation contains a flag value that is not defined for that operation, then the flag shall be ignored and an Improper_Flag_Use_Error should be logged.

10.7.8 Missing Blocks

If the data Destination detects that a Block of a Transfer is missing, it may re-issue the associated Clear_To_Send operation to request retransmission of the Block from a data Source that supports Out_Of_Order (see 5.1.1 and 8.2). Other actions to be taken if a Block is missing are beyond the scope of this standard.

Table 9 – Summary of logged errors

Name	Occurs in operation
Cksum_Error	all
Illegal_Blocksize_Error	CTS
Illegal_Bufsize_Error	CA, RC
Illegal_STU_Size_Error	Data
Improper_Flag_Use_Error	all
Invalid_D-id_Error	all with a non-zero D_id
Invalid_Key_Error	all except RC
Invalid_Mx_Error	Data
Invalid_Port_Error	all
Max_Retry_Occurance	End, DA, RC, RD, RS, RTR, RTS
Op_timeout_Occurance	End, DA, RC, RD, RS, RTR, RTS
Out_Of_Order_B_num	Data
Out_Of_Order_STU_Error	Data
Out_Of_Range_B_num_Error	CTS, Data, RS, RSR
Out_Of_Range_Bufx_Error	Data
Oversized_Offset_Error	Data
Slots_Exceeded_Error	all with Op ≥ x'06'
Undefined_Opcode_Error	not applicable
Undefined_Opcode_Value	not applicable
Unexpected_Opcode_Error	all except RC
Unexpected_Opcode_Value	all except RC
Unknown_EtherType_Error	RC

Operation abbreviations:
- CA = Connection_Answer
- CTS = Clear_To_Send
- DA = Disconnect_Answer
- RC = Request_Connection
- RD = Request_Disconnect
- RS = Request_State
- RSR = Request_State_Response
- RTR = Request_To_Receive
- RTS = Request_To_Send

Annex A
(normative)

Using lower layer protocols

This Scheduled Transfer Protocol (ST) may be used with a variety of lower layer and physical media protocols. Mappings to some of the more common protocols are specified below. This is not intended to be an all inclusive set of protocols, i.e., ST may be used with other LLPs than those listed. Specific items addressed by each mapping include:

– CCI information such as physical layer addresses,

– protocol data unit (PDU) size restrictions,

– and the mappings for the ST Control and Data Channels.

However, the methods used to pass this information between ST and the LLP are outside the scope of this standard.

For Request_Connection and Connection_Answer operations, connection control information (CCI) for the Virtual Connection being set up is passed to the specified LLP and may be stored in the Virtual Connection Descriptor (see figure 5). Examples of CCI parameters include, but are not limited to:

– LLP-specific destination address;

– LLP-specific source address;

– quality of service.

ST does not provide the initial CCI; it may come from the ULP or from another protocol. The CCI is not carried in the Schedule Header. However, an ST implementation would typically retain the CCI for further operations on the Port. Note that some situations, e.g., striping, may use other than the retained addresses.

A.1 HIPPI-6400-PH as the LLP

ANSI X3.xxx defines HIPPI-6400-PH, portions of which are repeated here as an aid to the reader. As shown in figure A.1, ST operations shall be carried over HIPPI-6400-PH with the first eight bytes of the Schedule Header occupying the last eight bytes of the HIPPI-6400-PH Header micropacket.

HIPPI-6400-PH specifies that its ULP (ST in this case), provide information to be used to generate the MAC header. The ULP provides the Destination address (D_ULA) in a Request_Connection operation, and may provide the Source address (S_ULA). In the corresponding Connection_Answer, the received S_ULA would be used as the D_ULA, and the ULP may provide the S_ULA value. (See HIPPI-6400-PH 5.3.1.) In ST these parameters are in the CCI. Included are:

– LLC/SNAP header with:
  – DSAP = x'AA' (SNAP);
  – SSAP = x'AA' (SNAP);
  – Ctl = x'03' (unnumbered packets);
  – Org = x'00', x'00', x'00' (generic packets);
  – EtherType = x'8181' (Scheduled Transfer).

– Destination physical address (D_ULA),

– optionally the Source physical address (S_ULA).

All ST Control operations shall be carried on HIPPI-6400-PH Virtual Channel VC0. Data operations shall use Virtual Channel 1, 2, or 3 as specified in the ST Data Channel Assignment flag bits (see 8.2) and carried in a Request_To_Send operation (see Q.2).

The buffers available in some HIPPI-6400-PH intermediate device implementations (e.g., for translators, routers, etc.), may limit the STU and Block sizes. These size restrictions shall be resolved with the Max_STU and Max_Block parameters (see 5.2.4 and 5.2.5).

M_len (in the HIPPI-6400-PH MAC Header), specifies the number of bytes following M_len, exclusive of any padding in the last micropacket. Hence, M_len will have the following values:

– M_len = 48 for Control operations without an optional payload (i.e., 48 = 8 byte IEEE 802.2 LLC/SNAP Header + 40-byte ST Schedule Header);

– M_len = 80 for Control operations with optional payload;

– M_len = (48 + number of user data payload bytes) for Data operations.

A.2 HIPPI-FP as the LLP

ANSI X3.210 defines HIPPI-FP, portions of which are repeated here as an aid to the reader. As shown in figure A.2, ST operations shall be carried over HIPPI-FP in the D2_Area. The HIPPI-FP D1_Area shall not be used. The HIPPI-FP D2_Offset shall be set to zero. Short bursts shall only be used at the end of a packet, i.e., short first burst is disallowed. Note that D2_Size = M_len + 16.

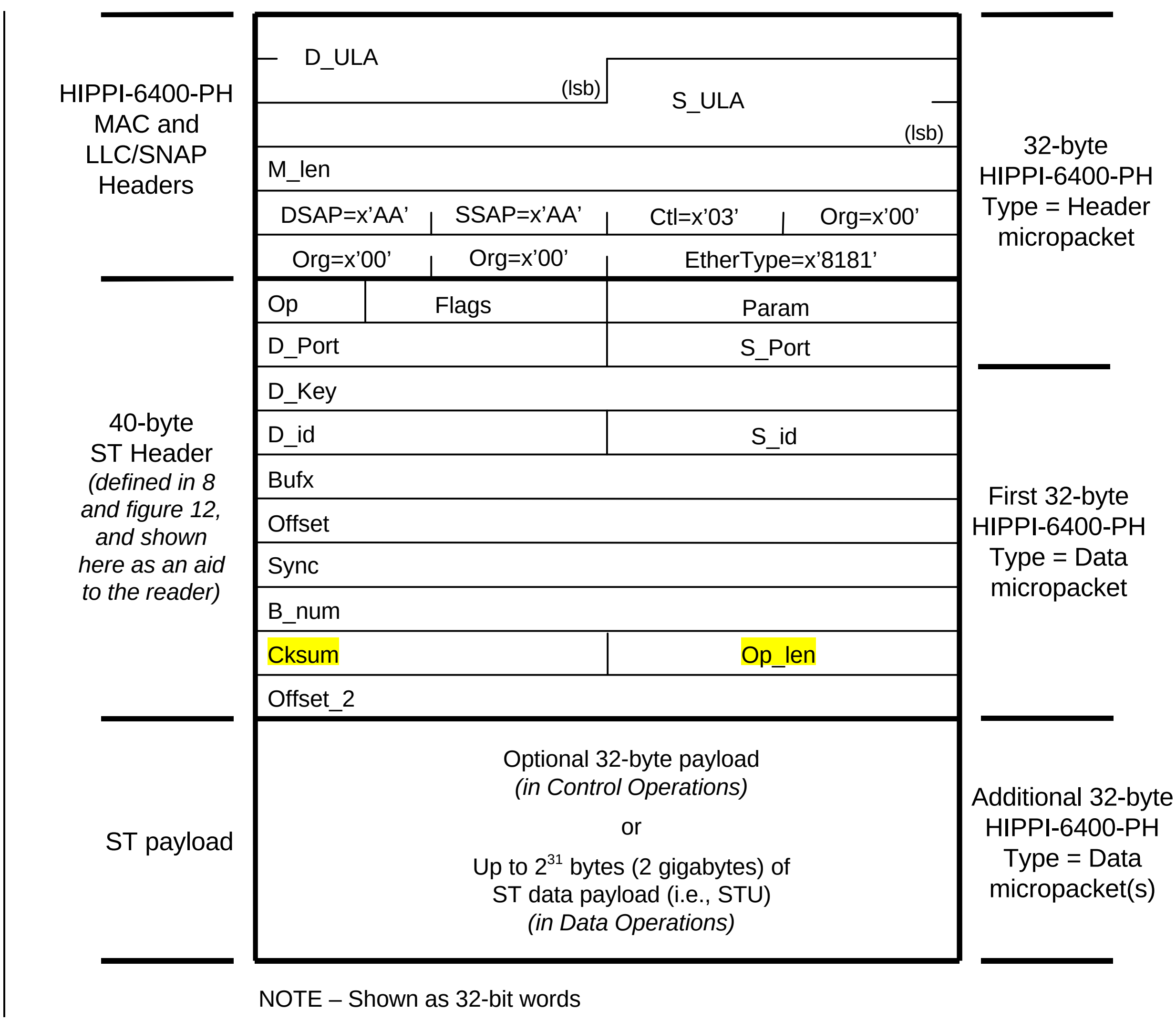


Figure A.1 – An ST operation carried in a HIPPI-6400-PH Message

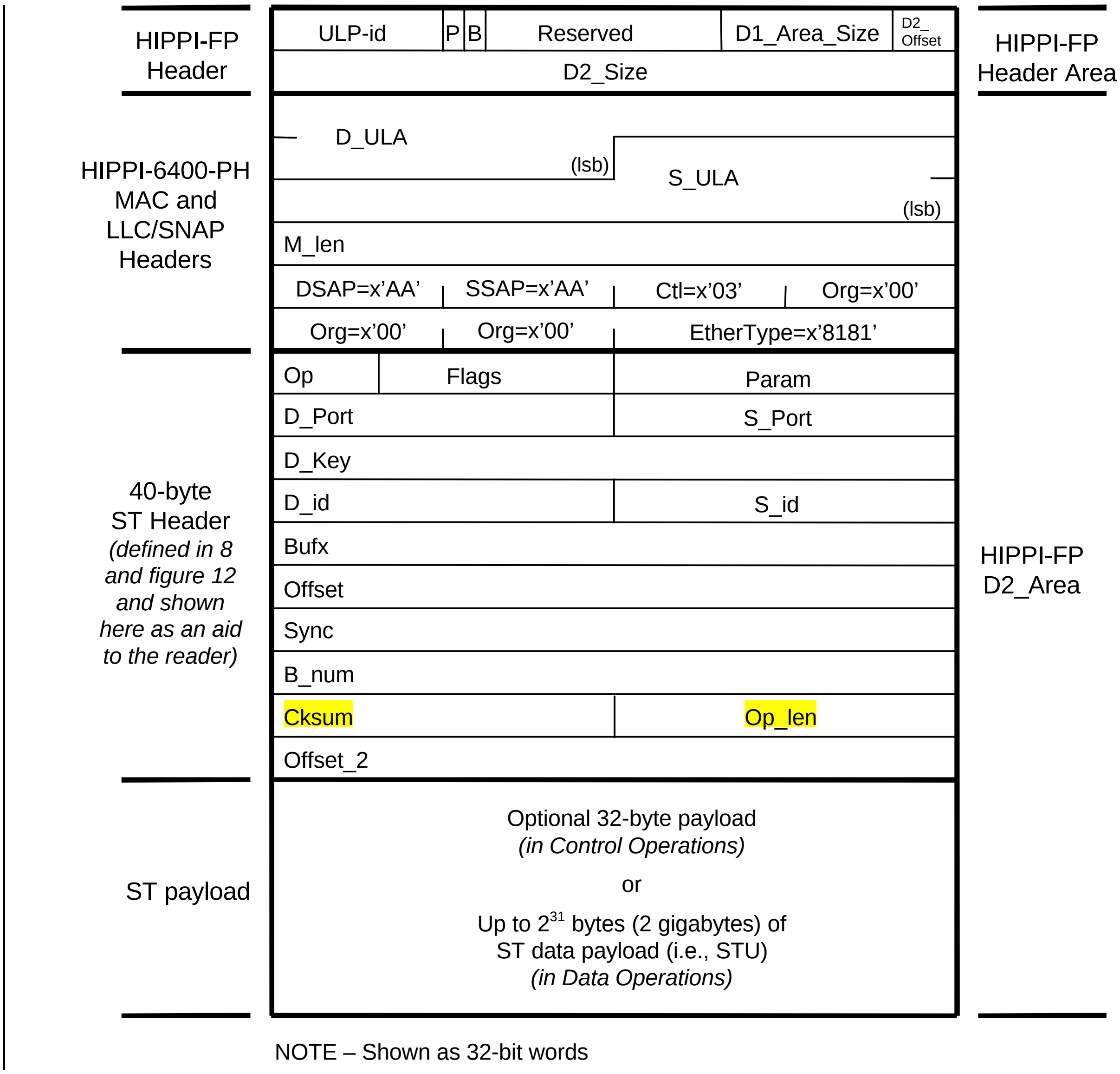


Figure A.2 – An ST operation carried in a HIPPI-FP packet

Open Issue – Should one, or both, Headers be in the HIPPI-FP D1_Area instead of the D2_Area? Another mapping may be added with the Header in the D1 area.

The HIPPI-6400-PH MAC and LLC/SNAP Headers are defined in ANSI X3.xxx, portions of which are repeated here as an aid to the reader. As shown in figure A.2, the MAC and LLC/SNAP headers shall precede the Schedule Header to facilitate translation to other protocols. The same CCI information as specified in A.1 above shall be used to create the MAC and LLC/SNAP headers. The following additional parameters shall be included in the CCI for use by the HIPPI-FP protocol:

– ULP-id = x'0C' signifying HIPPI-6400 Encapsulation,

– 12-bit Destination Addresses as specified by ANSI X3.222, High-Performance Parallel Interface - Physical Switch Control (HIPPI-SC).

All ST Control operations shall specify Virtual Channel VC0. Data operations shall specify Virtual Channel 1, 2, or 3 as specified in the ST Data Channel Assignment flag bits (see 8.2) and carried in a Request_To_Send operation (see Q.2). Note that HIPPI-FP and ANSI X3.183, High-Performance Parallel Interface – Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH), also known as HIPPI-800, do not provide multiple channels. Therefore, all of the Control and Data operations share the HIPPI-800 physical link, and a long Data operation can block delivery of Control operations until the Data operation completes. The Virtual Channel specifications are needed when going from HIPPI-800 to HIPPI-6400.

All ST Control operations shall specify Virtual Channel VC0. Data operations shall specify Virtual Channel 1, 2, or 3 as specified in the ST Data Channel Assignment flag bits (see 8.2) and carried in a Request_To_Send operation (see Q.2). Note that HIPPI-FP and ANSI X3.183, High-Performance Parallel Interface - Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH), also known as HIPPI-800, do not provide multiple channels. Therefore, all of the Control and Data operations share the HIPPI-800 physical link, and a long Data operation can block delivery of Control operations until the Data operation completes. The Virtual Channel specifications are needed when going from HIPPI-800 to HIPPI-6400.

The buffers available in some HIPPI-FP intermediate device implementations (e.g., for translators, routers, etc.), may limit the STU and Block sizes. These size restrictions shall be resolved with the Max_STU and Max_Block parameters (see 5.2.4 and 5.2.5).

A.3 Ethernet as the LLP

Figure A.3 shows a 40-byte ST header immediately following the 14-byte Ethernet 802.3 MAC header and 8-byte 802.1 SNAP header. An 802.3 MAC header is needed because the ST header does not contain a length field, and because the number of non-pad bytes in an Ethernet frame cannot be inferred from the physical frame size. The SNAP header is required because the ST header does not contain an EtherType. The SNAP header values shall be identical to those defined for use with HIPPI-6400-PH as the LLP (see A.1), i.e.,:

– DSAP = x'AA' (SNAP);
– SSAP = x'AA' (SNAP);
– Ctl = x'03' (unnumbered packets);
– Org = x'00', x'00', x'00' (generic packets);
– EtherType = x'8181' (Scheduled Transfer).

The payload bytes are optional. If the ST header contains a Control operation, then the payload shall be either zero bytes or 32 bytes (see 4.2). If the ST header contains a Data operation, then the payload can be any size up to 1024 bytes.

Ethernet frames must be an even number of bytes and must also satisfy a minimum length requirement. In conventional implementations these constraints are satisfied by logic in low-level device drivers as well as physical-layer hardware. Since ST is an upper-layer client of the physical layer, the padding details are not described here.

The length of an ST message on Ethernet is determined from the 16-bit 802.3 length field which specifies the number of bytes following the length field exclusive of any padding. Possible values for the length are:

– len = 48 for Control operations without an optional payload.

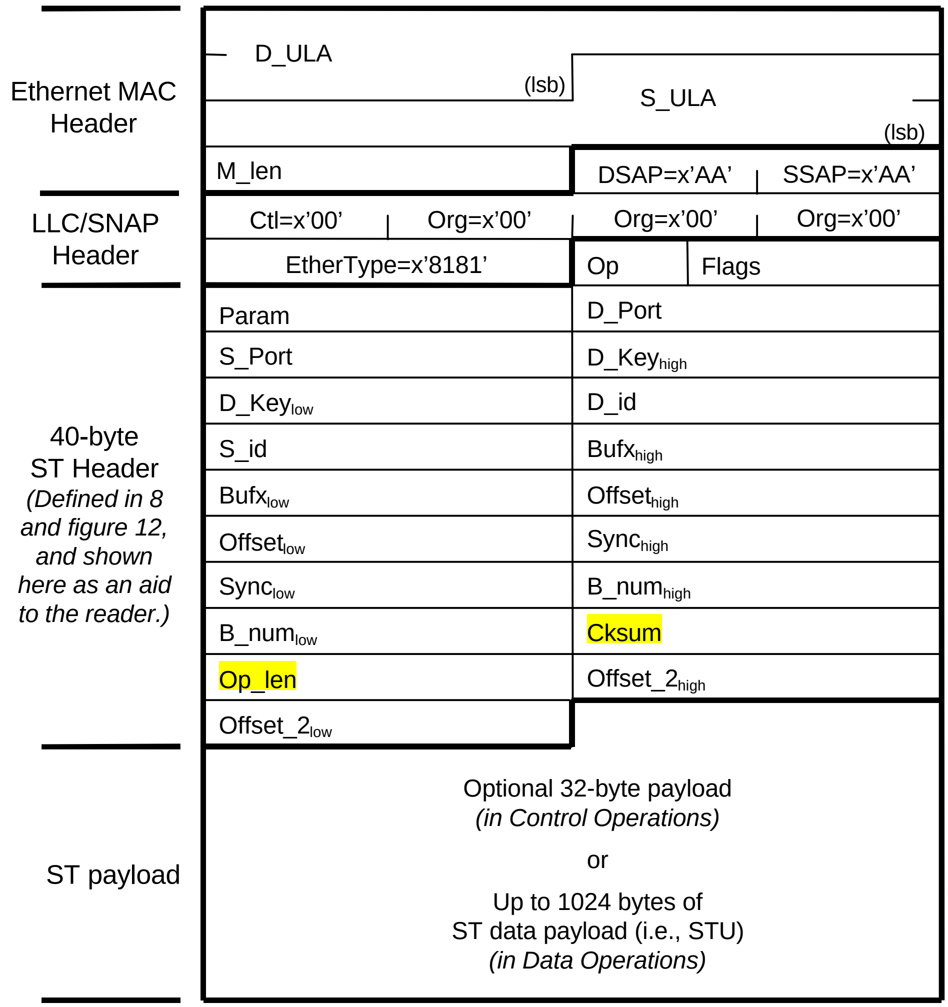
– len = 80 for Control operations with payload

– len = 48 + (size of payload) for Data operations

The EtherTypes used with ST on Ethernet shall be as specified in 5.2.7.

Ethernet does not provide virtual circuits or virtual channels, therefore both Control and Data operations are queued and processed in FIFO order. Control operations and Data operations may be interleaved: there is no requirement for all the STUs of a multi-STU Block to be transmitted contiguously on the medium.

ST requires in-order transmission and delivery of STUs within a Block. It is expected that many implementations will handle out-of-order Blocks. If an Ethernet environment cannot preserve STU ordering, then the Blocksize (see 6.2.5) should be set to 1024 bytes, making use of out-of-order Block processing.



NOTE – Shown as 32-bit words

Figure A.3 – An ST operation carried in an Ethernet packet

A.4 ATM LAN Emulation as the LLP

Open Issue – The contents of this clause may be based on Robert Hyerle's proposal.

A.5 Fibre Channel as the LLP

Open Issue – Jerry Leitherer of Genroco has an action item to draft text for ST over Fibre Channel.

Annex B
(informative)

ST striping

B.1 Striping principles

ST is capable of supporting multiple physical interfaces for a single Transfer (see figure B.1). This striping capability may be of benefit when a single interface is not able to support required data rates. It may be especially useful where data is moved from many slower interfaces to a single faster interface or vice-versa. It may also be used with multiple interfaces at both the Source and Destination. Mechanisms to set up, select, and control the underlying physical interfaces are beyond the scope of this standard.

The Block is the basic striping unit. Each Block contains sufficient information to completely identify an individual Transfer and the Block's location within the Transfer. The only difference between striped and non-striped operation is the selection of port MAC addresses to allow concurrent data movement. Striping is not done on an STU basis because striped STUs can not be guaranteed to be delivered in-order as required by ST.

There are a few conventions that should be followed to facilitate striping:

– Block sizes (when striping is desirable) must be small enough to support concurrency and allow each channel to have at least one Block to send.

– Sufficient Clear_To_Send operations should be kept outstanding by a data receiver to allow concurrent Data operations.

– The interface adapter(s) must be capable of handling multiple Blocks simultaneously. This may require communication between interfaces (or their software drivers) within a system.

– The return physical address (e.g., Source ULA), for each operation is specified by the LLP source address for that operation. ST implementers should not assume that the source LLP address for a given Port will remain constant.

– The Destination must signify that it supports delivery of Blocks in any order (i.e., Out_of_Order = 1, see 8.2) during the Virtual Connection setup.

B.2 Many-to-one striping

Part a of figure B.1 shows using a number of lower-throughput interfaces, aggregated together, to communicate with one higher-throughput interface (using a translator or bridge). Striping the lower-throughput interfaces together can allow legacy systems to communicate quickly over newer network infrastructures. In this case, action to implement striping is required only on the side of the lower-throughput interface.

After the Ports are assigned, data movement is initiated with a Request_To_Send operation. A Request_To_Send_Response will be received, either as a discrete message or as part of a Clear_To_Send. As Clear_To_Send operations are received, the system with multiple lower-throughput ports can move a Block of data for each Clear_To_Send received. As many Blocks can be in transit concurrently as there are ports to carry them and Clear_To_Send operations authorizing them.

The system receiving these Blocks processes them normally, placing them into memory as their Bufx and Offset values dictate.

B.3 One-to-many striping

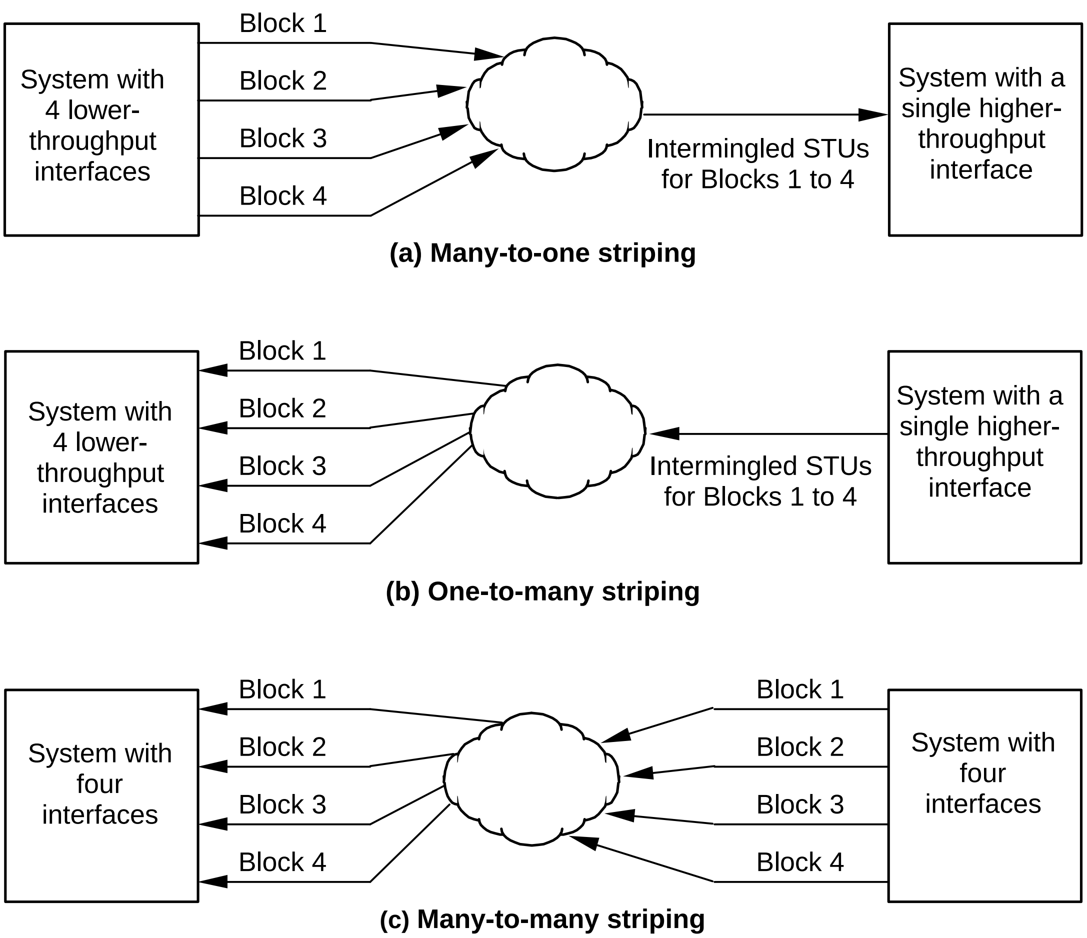
Part b of figure B.1 shows how Transfers made from one higher-throughput interface can also be spread across more than one lower-throughput interface without any special action on the part of the higher-throughput system.

After the Ports are assigned, the Transfer is initiated with a Request_To_Send operation from the higher-throughput interface. The lower-throughput interface that has done the Port assignment will return a Request_To_Send_Response, either as a discrete operation or as part of a Clear_To_Send. Each Clear_To_Send issued should be sent from the interface desiring the data.

An alternative is to send all of the Clear_To_Send operations from a single interface and substitute the desired physical return address (e.g., Source physical address) for the Clear_To_Send's Source physical address (making it appear that the Clear_To_Send's Source physical address was generated by the interface desiring the data). Subsequent Data operations may then be done concurrently and will use a Source physical address from the Clear_To_Send operation as the Destination physical address. Using this substitution method in combination with a dedicated control channel may also prevent or reduce blocking effects where the underlying physical medium suffers from high latency.

B.4 Many-to-many striping

Part c of figure B.1 shows many-to-many striping as the combination of the one-to-many and many-to-one striping. The system receiving data indicates its desire to receive in a striped fashion by issuing multiple Clear_To_Send operations with differing return interface addresses. The system sending data chooses to stripe by sending from multiple interfaces that are capable of reaching the proper destination.



NOTE: A Clear_To_Send for each Block is sent in the reverse direction on the same path that each Block traverses (or is made to appear that way) for figure B.1 (a-c).

Figure B.1 – ST Striping Configurations

Annex C
(informative)

Scheduled Transfer Protocol examples

NOTE – This annex has not been updated for some time, and hence is quite out of date, e.g., names, functions, etc.. Rather than leave it in place and possibly confuse readers, it has been removed until it can be brought up to date. Jim Pinkerton has offered to draft text based on his work at SGI.

Annex D
(informative)

State tables

Open Issue – Jeffrey Chung of SGI has an action item to provide the state tables. Note that the state tables are informative rather than normative.