



LINX Protocols

Document Version 20

ENEA

LINX Protocols

Copyright

Copyright © Enea Software AB 2009.

Disclaimer

The information in this User Documentation is subject to change without notice, and should not be construed as a commitment of Enea Software AB.

Trademarks

Enea®, Enea OSE®, and Polyhedra® are the registered trademarks of Enea AB and its subsidiaries. Enea OSE@ck, Enea OSE® Epsilon, Enea® Element, Enea® Optima, Enea® LINX, Enea® Accelerator, Polyhedra® FlashLite, Enea® dSPEED, Accelerating Network Convergence™, Device Software Optimized™, and Embedded for Leaders™ are unregistered trademarks of Enea AB or its subsidiaries. Any other company, product or service names mentioned in this document are the registered or unregistered trademarks of their respective owner.

Table of Contents

1 - Introduction to the LINX Protocol	5
1.1 - Abstract	5
1.2 - Document Revision History	5
1.3 - Definitions and Acronyms	5
2 - Overview of the LINX Protocol	7
3 - LINX Protocols	9
3.1 - LINX Link Creation and Initialization	9
3.2 - RLNH Feature Negotiation	9
3.3 - Publication of Names	9
3.4 - Remote Name Lookup	10
3.5 - Link Supervision	10
3.6 - Protocol Messages	10
3.6.1 - RLNH_INIT	10
3.6.2 - RLNH_INIT_REPLY	10
3.6.3 - RLNH_PUBLISH	11
3.6.4 - RLNH_QUERY_NAME	11
3.6.5 - RLNH_UNPUBLISH	12
3.6.6 - RLNH_UNPUBLISH_ACK	12
3.6.7 - RLNH_PUBLISH_PEER	12
4 - Enea LINX Connection Manager Protocols	14
4.1 - Connection Establishment	14
4.2 - Reliable Message Passing	14
4.3 - Connection Supervision	14
5 - Enea LINX Ethernet Connection Manager	15
5.1 - Protocol Descriptions	15
5.1.1 - Enea LINX Ethernet Connection Manager Headers	15
5.1.2 - Enea LINX Connect Protocol	16
5.1.3 - Enea LINX User Data Protocol	20
5.1.4 - Enea LINX Reliability Protocol	21
5.1.5 - Enea LINX Connection Supervision Protocol	23
5.2 - LINX Discovery Daemon	24
5.2.1 - Linxdisc protocol	25
6 - Enea LINX Point-To-Point (PTP) Connection Manager	27
6.1 - Protocol Description	27
6.1.1 - Enea LINX Point-To-Point Connection Manager Headers	27
6.1.2 - PTP Connection Supervision Protocol	30
7 - Enea LINX TCP Connection Manager	31
7.1 - TCP CM Protocol Descriptions	31
7.1.1 - TCP Connection Manager Headers	31
7.1.2 - TCP CM Connection Supervision Protocol	32
8 - Enea LINX Gateway Protocol	33
8.1 - Gateway Protocol Description	33
8.1.1 - Generic Request/Reply Header	34
8.1.2 - Interface Request/Reply Payload	34
8.1.3 - Create Request/Reply Payload	34
8.1.4 - Destroy Request/Reply Payload	35
8.1.5 - Send Request/Reply Payload	35
8.1.6 - Receive Request/Reply Payload	35
8.1.7 - Hunt Request/Reply Payload	36
8.1.8 - Attach Request/Reply Payload	36
8.1.9 - Detach Request/Reply Payload	37
8.1.10 - Name Request/Reply Payload	37

9 - Enea LINX Shared Memory Connection Manager	38
9.1 - Protocol Description	38
9.1.1 - Shared Memory Protocol Headers	38
9.1.2 - SHM Connection Protocol Description	39
10 - Enea LINX RapidIO Connection Manager	41
10.1 - Protocol Description	41
10.1.1 - LINX RapidIO Connection Establishment Algorithm	41
10.1.2 - Enea LINX RapidIO User Data Protocol	44
10.1.3 - Enea LINX RapidIO Connection Supervision Protocol	47
Index	49

1. Introduction to the LINX Protocol

1.1 Abstract

This document describes the Enea® LINX protocols, used in Enea LINX. See the Revision History section (and document header line) for the version of this document.

LINX is a distributed communication protocol stack for transparent inter node and inter process communication for a heterogeneous mix of systems.

1.2 Document Revision History

Revision	Author	Date	Status and Description
20	wivo	2010-02-16	Added riocm protocol description.
19	tomk	2009-12-03	Added shmcm protocol description.
18	lejo	2009-03-23	Converted document from xhtml to Docbook XML.
17	tomk	2009-03-09	Added LINX Gateway protocol.
16	wivo	2008-07-10	Updated TCP CM protocol with OOB.
15	wivo	2008-05-13	Updated Ethernet protocol with OOB.
14	debu	2007-11-05	Document updates in RLNH and Ethernet protocol.
13	debu	2007-09-25	Updated Linxdisc protocol description.
12	mwal	2007-09-24	Added feature negotiation. Increased rlnh version to 2 and ethcm version to 3.
11	lejo,wivo	2007-09-20	Added TCP CM ver.2
10	zalpa,lejo	2007-08-29	Approved. Added PTP CM ver.1
9	lejo	2006-10-25	Added copyright on front page. Cleaned out prerelease entries of document history.
8	lejo	2006-10-13	Converted document from Word to XHTML.
7	jonj	2006-09-13	Added reserved field in UDATA Header, Fixed bug in MAIN header.
6	jonj	2006-09-11	Fixed a few bugs, improved RLNH protocol description.
5	jonj	2006-08-05	Approved. Updated after review (internal reference ida 010209).

1.3 Definitions and Acronyms

Definitions

A	The active (initiating) side in a protocol exchange.
B	The passive (responding) side in a protocol exchange.
CM	Connection Manager, the entity implementing the transport layer of the Enea LINX protocol.

Endpoint	A (part of) an application that uses the Enea LINUX messaging services.
Connection	An association between Connection Managers.
Connection ID	A key used by Ethernet Connection Managers to quickly lookup the destination of an incoming packet. Connection ID based lookup is much efficient than MAC-address based lookup.
Connection Manager	A Connection Manager provides reliable communication for message passing. The connection layer roughly corresponds to layer four, the transport layer, in the OSI model.

Abbreviations

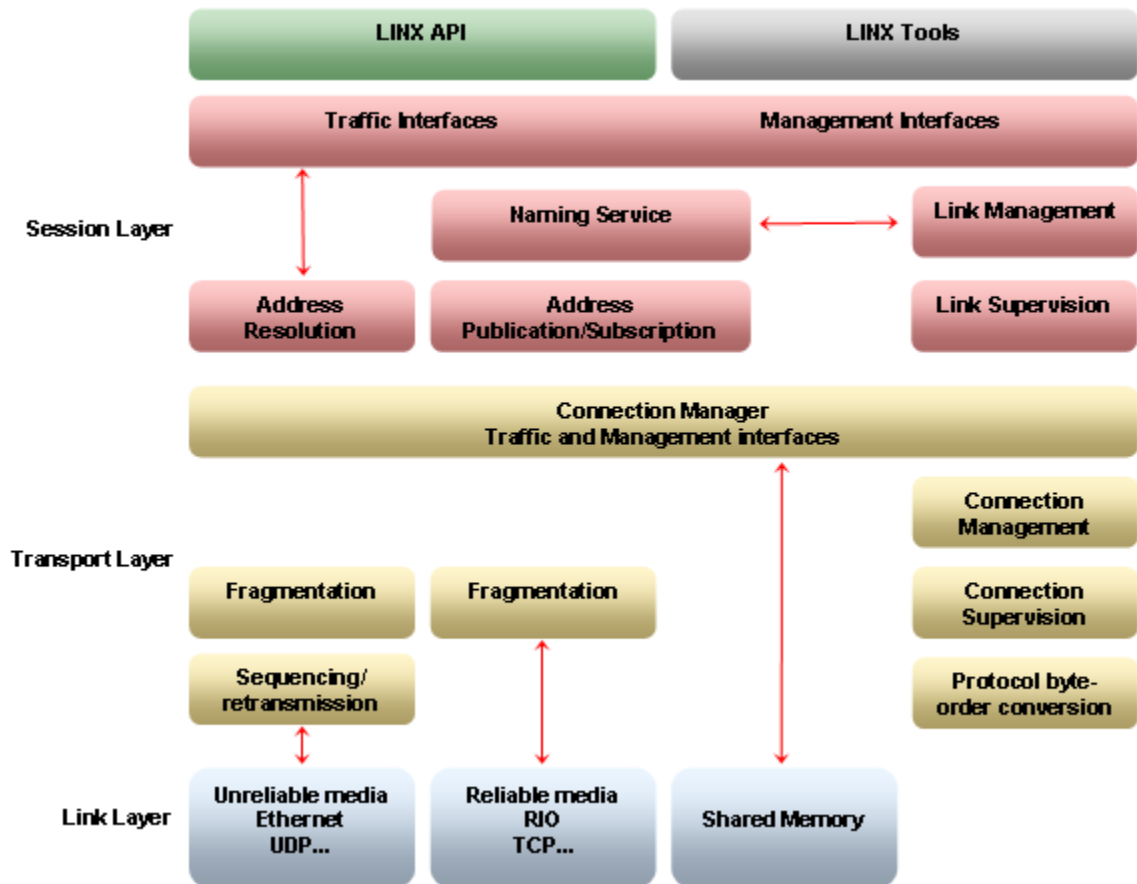
IPC Inter Process Communication.

PDU Protocol Data Unit..

2. Overview of the LINX Protocol

Enea LINX is an open technology for distributed system IPC which is platform and interconnect independent, scales well to large systems with any topology, but that still has the performance needed for high traffic bearing components of the system. It is based on a transparent message passing method.

Figure 2.1 LINX Architecture



Enea LINX provides a solution for inter process communication for the growing class of heterogeneous systems using a mixture of operating systems, CPUs, microcontrollers DSPs and media interconnects such as shared memory, RapidIO, Gigabit Ethernet or network stacks. Architectures like this poses obvious problems, endpoints on one CPU typically uses the IPC mechanism native to that particular platform and they are seldom usable on platform running other OSes. For distributed IPC other methods, such as TCP/IP, must be used but that comes with rather high overhead and TCP/IP stacks may not be available on small systems like DSPs. Enea LINX solves the problem since it can be used as the sole IPC mechanism for local and remote communication in the entire heterogeneous distributed system.

The Enea LINX protocol stack has two layers - the RLNH and the Connection Manager, or CM, layers. RLNH corresponds to the session layer in the OSI model and implements IPC functions including methods to look up endpoints by name and to supervise to get asynchronous notifications if they die. The Connection Manager layer corresponds to the transport layer in the OSI model and implements reliable in order transmission of arbitrarily sized messages over any media.

RLNH is responsible for resolving remote endpoint names, and for setting up and removing local representations of remote endpoints. The RLNH layer provides translation of endpoint OS IDs from the source to the destination system. It also handles the interaction with the local OS and applications that use the Enea LINX messaging services. RLNH consists of a common, OS independent protocol module and an OS adaptation layer that handles OS specific interactions.

The Connection Manager provides a reliable transport mechanism for RLNH. When the media is unreliable, such as Ethernet, or have other quirks the Connection Manager must implement means like flow control, retransmission of dropped packets, peer supervision, and becomes much more complex. On reliable media, such as shared memory or RapidIO, Connection Managers can be quite simple.

The rest of this document contains a detailed description of the protocols used by the RLNH and the CM layers of Enea LINX. Chapter 3 describes the RLNH protocol. Chapter 4 describes features shared by all Connection Managers, i.e. what RLNH requires from a Connection Manager. The description is intentionally kept on a high level since the functionality actually implemented in any instance of a CM depends very much on the properties of the underlying media. It is the intention that when Connection Managers for new media are implemented the protocols will be described in the following chapters. Finally Chapter 5 describes version two of the Ethernet Connection Manager protocol.

The table below shows how protocol headers are described. PDUs are sent in network byte order (e.g. big endian). Bits and bytes are numbered in the order they are transmitted on the media. For example:

Table 2.1 Protocol Message Legend

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
One								Two				Three				Reserved															
Four																															

The first row shows bit numbers, the second row byte numbers for those more comfortable with that view, and the last row example protocol fields. In the header above, the fields in the header are sent in order: one, two, three, reserved, four.

3. LINX Protocols

The LINX protocol (previously called RLNH protocol) is designed to be light-weight and efficient. The RLNH-to-RLNH control PDUs are described in detail below. The required overhead associated with user signal transmission is not carried in any dedicated RLNH PDU. Instead it is passed as arguments along with the signal data to the Connection Manager layer, where an optimized transmission scheme and message layout can be implemented based on knowledge of the underlying media and/or protocols. In particular the source and destination link addresses are sent this way, the addresses identifies the sending and receiving endpoints respectively. LINX protocol messages are sent with source and destination link addresses set to zero (0).

3.1 LINX Link Creation and Initialization

The Connection Manager is initialized by RLNH. It is responsible for providing reliable, in-order delivery of messages and for notifying RLNH when the connection to the peer becomes available / unavailable. An RLNH link is created maps a unique name to a Connection Manager object.

As soon as the Connection Manager has indicated that the connection is up, RLNH transmits an RLNH_INIT message to the peer carrying its protocol version number. Upon receiving this message, the peer responds with an RLNH_INIT_REPLY message indicating whether it supports the given protocol version or not. When remote and local RLNH versions differ the lowest of the two versions are used by both peers. If the message exchange has been successfully completed, RLNH is ready to provide messaging services for the link name. The RLNH protocol is stateless after this point.

3.2 RLNH Feature Negotiation

The RLNH Feat_neg_string is sent once by both peers to negotiate what features enable. It is contained in the INIT_REPLY message. The RLNH Feat_neg_string contain a string with all feature names and corresponding arguments. Only the features used by both peers (the intersection) are enabled. All the features supported by none or one peer (the complement) are disabled by both peers. An optional argument can be specified and It is up to each feature how the argument is used and how the negotiation of the argument is performed.

3.3 Publication of Names

RLNH assigns each endpoint a link address identifier. The association between the name of an endpoint and the link address that shall be used to refer to it is published to the peer in an RLNH_PUBLISH message. RLNH_PUBLISH is always the first RLNH message transmitted when a new endpoint starts using the link.

Upon receiving an RLNH_PUBLISH message, RLNH creates a local representation of the remote name. Local applications can communicate transparently with such a representation, but the signals are in reality forwarded by RLNH to the peer system where the real destination endpoint resides (and the destination will receive the signals from a representation of the true sender).

The link addresses of source and destination for each signal are given to the Connection Manager for transmission along with the signal data and delivered to the peer RLNH. The link addressing scheme is designed to enable O(1) translation between link addresses and their corresponding local OS IDs.

3.4 Remote Name Lookup

When a hunt call is performed for the name of a remote endpoint, the request is passed on by RLNH to the peer as a RLNH_QUERY_NAME message. If the hunter has not used the link before, an RLNH_PUBLISH message is sent first.

Upon receiving a RLNH_QUERY_NAME message, RLNH resolves the requested endpoint name locally and returns an RLNH_PUBLISH message when it has been found and assigned a link address. As described above, this triggers the creation of a representation at the remote node. This in turn resolves the hunt call - the hunter is provided with the OS ID of the representation.

3.5 Link Supervision

RLNH supervises all endpoints that use the link. When a published name is terminated, a RLNH_UNPUBLISH message with its link address is sent to the peer.

When a RLNH_UNPUBLISH message is received, RLNH removes its local representation of the endpoint referred to by the given link address. When RLNH no longer associates the link address with any resources, it returns a RLNH_UNPUBLISH_ACK message to the peer.

The link address can be reused after a RLNH_UNPUBLISH_ACK message.

3.6 Protocol Messages

3.6.1 RLNH_INIT

The RLNH_INIT message initiates link establishment between two peers. It is sent when the Connection Manager indicates to RLNH that the connection is up.

Table 3.1 RLNH Init Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
Version																															

Reserved Reserved for future use, must be 0.

Type Message type. The value of RLNH_INIT is 5.

Version The RLNH protocol version. The current version is 2.

3.6.2 RLNH_INIT_REPLY

During link set up, RLNH responds to the RLNH_INIT message by sending an RLNH_INIT_REPLY message. The status field indicates whether the protocol is supported or not. The Feat_neg_string tells the remote RLNH what features the local RLNH supports.

Table 3.2 RLNH Init Reply Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															

Status	
Feat_neg_string (variable length, null-terminated string)	
Reserved	Reserved for future use, must be 0.
Type	Message type. The value of RLNH_INIT_REPLY is 6.
Status	Status code indicating whether the protocol version received in the RLNH_INIT message is supported (0) or not (1).
Feat_neg_string	String containing feature name and argument pairs. Example: "feature1:arg1,feature2:arg2\0".

3.6.3 RLNH_PUBLISH

The RLNH_PUBLISH message publishes an association between an endpoint name and the link address that shall be used to refer to it in subsequent messaging. Upon receiving an RLNH_PUBLISH message, RLNH creates a local representation of the remote name. This resolves any pending hunt calls for link_name/remote_name.

Table 3.3 RLNH Publish Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
Linkaddr																															
Name (variable length, null-terminated string)																															

Reserved	Reserved for future use, must be 0.
Type	Message type. The value of RLNH_PUBLISH is 2.
Linkaddr	The link address being published.
Name	The name being published..

3.6.4 RLNH_QUERY_NAME

The RLNH_QUERY_NAME message is sent in order to resolve a remote name. An RLNH_PUBLISH message will be sent in response when the name has been found and assigned a link address by the peer.

Table 3.4 RLNH Query Name Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
src_linkaddr																															
Name (variable length, null-terminated string)																															

Reserved	Reserved for future use, must be 0.
Type	Message type. The value of RLNH_QUERY_NAME is 1.

src_linkaddr Link address of the endpoint that issued the query.

Name The name to be looked up.

3.6.5 RLNH_UNPUBLISH

The RLNH_UNPUBLISH message tells the remote RLNH that the endpoint previously assigned the given link-address has been closed.

Table 3.5 RLNH Unpublish Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
Linkaddr																															

Reserved Reserved for future use, must be 0.

Type Message type. The value of RLNH_UNPUBLISH is 3.

Linkaddr The address of the closed endpoint.

3.6.6 RLNH_UNPUBLISH_ACK

The RLNH_UNPUBLISH_ACK message tells the remote RLNH that all associations regarding an unpublished link address have been terminated. This indicates to the peer that it is ok to reuse the link address.

Table 3.6 RLNH Unpublish Ack Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
Linkaddr																															

Reserved Reserved for future use, must be 0.

Type Message type. The value of RLNH_UNPUBLISH_ACK is 4.

Linkaddr The link address of the unpublished endpoint.

3.6.7 RLNH_PUBLISH_PEER

When a remote LINX endpoint is used as the sender in a send_w_sender() function call and the receiver exists on the same node as the local LINX endpoint, the remote LINX endpoint is published as a remote sender.

Table 3.7 RLNH Publish Peer Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved																Type															
Linkaddr																															
Peer_linkaddr																															

Reserved	Reserved for future use, must be 0.
Type	Message type. The value of RLNH_PUBLISH_PEER is 5.
Linkaddr	The link address being published.
Peer_linkaddr	The link address of the endpoint that previously published it self on the current link.

4. Enea LINX Connection Manager Protocols

This chapter describes in general terms the functionality a Connection Manager must provide.

A Connection Manager shall hide details of the underlying media, e.g. addressing, general media properties, how to go about to establish connections, media aggregation, media redundancy and so on. A Connection Manager shall support creation of associations, known as connections that are suitable for reliable message passing of arbitrarily sized messages. A Connection Manager must not rely on implicit connection supervision since this can cause long delay between peer failure and detection if the link is idle. A message accepted by a Connection Manager must be delivered to its destination.

If, for any reason, a Connection Manager is unable to deliver a message RLNH must be notified and the connection restarted since its state at this point is inconsistent.

Since Enea LINX runs on systems with very differing size, the Connection Manager protocol must be scalable. A particular implementation may ignore this requirement either because the underlying media doesn't support scalability or that a non-scalable implementation has been chosen. However, interfaces to the Connection Manager and protocol design when the media supports big configuration shall not make design decisions that prevents the system to grow to big configurations should the need arise.

Enea LINX must be able to coexist with other protocols when sharing media.

4.1 Connection Establishment

Creation of connections always requires some sort of hand shake protocol to ensure that the Connection Manager at the endpoints agrees on communication parameters and the properties of the media.

4.2 Reliable Message Passing

A channel suitable for reliable message passing must appear to have the following properties:

- Messages are delivered in the order they are sent.
- Messages can be of arbitrary size.
- Messages are never lost.
- The channel has infinite bandwidth.

Although no medium has all these properties some come rather close and others simulate them by implementing functions like fragmentation of messages larger than the media can handle, retransmission of lost messages, and flow control i.e. don't send more than the other side can consume. If a Connection Manager is unable to continue deliver messages according to these rules the connection must be reset and a notification sent to RLNH.

4.3 Connection Supervision

A distributed IPC mechanism should support means for applications to be informed when a remote endpoint fails. Connection Managers are responsible for detecting when the media fails to deliver messages and when the host where a remote endpoint runs has crashed. There are of course other reasons an endpoint might fail to respond but in those situations the communication link continues to function and higher layers in the LINX architecture manages supervision.

5. Enea LINX Ethernet Connection Manager

This chapter describes version 3 of the Enea LINX Ethernet Connection Manager Protocol.

5.1 Protocol Descriptions

Enea LINX PDUs are stacked in front of, possible, user data to form an Enea LINX Ethernet packet. All PDUs contain a field next header which contain the protocol number of following headers or the value -1 (1111b) if this is the last PDU. All headers except Enea LINX main header are optional. Everything from the transmit() down call, including possible control plane signaling, from the RLNH layer is sent reliably as user data. The first field in all headers is the next header field, having this field in the same place simplifies implementation and speeds up processing.

If a malformed packet is received the Ethernet Connection Manager resets the connection and informs RLNH.

When the Ethernet Connection Manager encounters problems which prevents delivery of a message or part of a message it must reset the connection. Notification of RLNH is implicit in the Ethernet CM, when the peer replies with RESET or, if the peer has crashed, the Connection Supervision timer fires.

5.1.1 Enea LINX Ethernet Connection Manager Headers

Version 3 of the Enea LINX Ethernet Connection Manager protocol defines these headers.

Table 5.1 Ethernet Connection Manager Protocol Headers

Protocol number	Value	Definition
ETHCM_MAIN		Main header sent first in all Enea LINX packets.
ETHCM_CONN		Connect header. Used to establish and tear down connections.
ETHCM_UDATA		User data header. All messages generated outside the Connection Manager are sent as UDATA.
ETHCM_FRAG		Fragment header. Messages bigger than the MTU are sent fragmented and PDUs following the first carries the ETHCM_FRAG header instead of ETHCM_UDATA.
ETHCM_ACK		Reliability header. Carries seqno and ackno. ACK doubles as empty acknowledge PDU and ACK-request PDU, in sliding window management and connection supervision.
ETHCM_NACK		Request retransmission of one or more packets.
ETHCM_NONE		Indicates that the current header is the last in the PDU.

5.1.1.1 ETHCM_MAIN Header

The ETHCM_MAIN header is sent first in all Enea LINX PDU's. It carries protocol version number, connection id, and packet size. Connection ID is negotiated when a connection is establish and is used to lookup the destination for incoming packets.

Table 5.2 ETHCM Main header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				Ver			Res		Conn_ID							R	Packet size														

Next	Next header, the protocol number of the following Enea LINX header or 1111b if last header.
Ver	Enea LINX Ethernet Connection Manager protocol version. Version 3 decimal is currently used, 0 is illegal.
Res	Reserved for future use, must be 0.
Conn_ID	A key representing the connection, used for fast identification of destination of incoming packets.
R	Reserved for future use, must be 0.
Packet size	Total packet size in bytes including this and following headers.

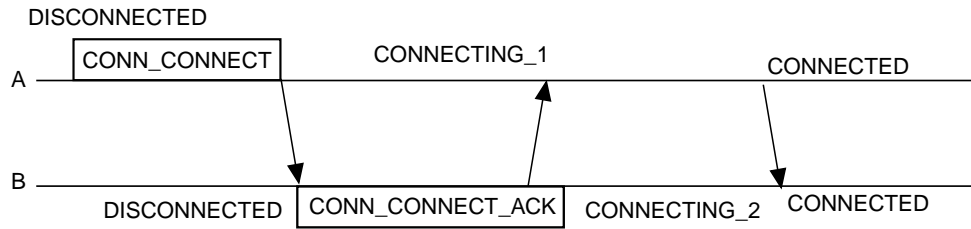
5.1.2 Enea LINX Connect Protocol

The Enea LINX Connect Protocol is used to establish a connection on the Connection Manager level between two peers A and B. A Connection Manager will only try to establish a connection or accept connection attempt from a peer if it has been explicitly configured to do. After configuration a CM will maintain connection with the peer until explicitly told to destroy the connection or an un-recoverable error occurs.

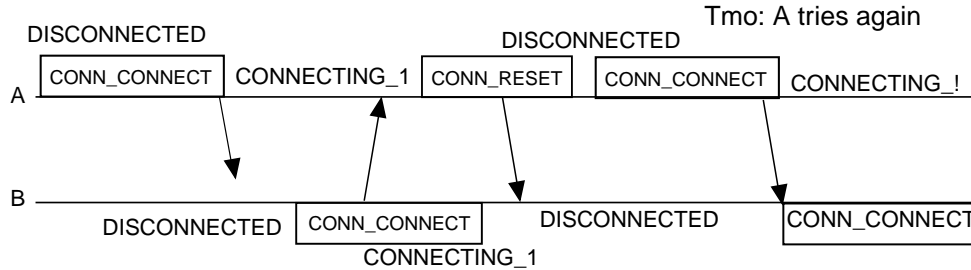
If a Connect-message is received, with a version number different from 2, the Ethernet CM refuses to connect.

The Connect Protocol determines Connection IDs to be used for this connection. Connection IDs are small keys used by receivers to quickly lookup a packets destination. Each side selects the Connection ID to be used by the peer when sending packets over the connection. The peer saves the ID and will use it for all future communication. If a node has a lot of connections it may run out of available Connection IDs. In this case the node sends Connection ID 0, which means no Connection ID and reverts to the slower MAC-addressing mode to determine destination for incoming packets.

A window size options may be sent in the CONNECT-header to indicate how the sender configuration deviates from the default. Deviations from default values are configured per connection in the `create_conn()`-call.

Figure 5.1 Successful Connect

Below, A starts first and tries to connect. B is not active, so the PDU is lost. A tries again..



Above, when B tries to connect to A, A is in the wrong state and sends RESET to synchronize

5.1.2.1 Connect Protocol

In the following protocol description the side initiating the connection is called A and the side responding to the request is called B. All protocol transitions are supervised by a timer, if the timer fires before the next step in the protocol have been completed the state machine reverts to state disconnected.

The Connect Protocol is symmetrical, there is no master and both sides try to initiate the connection. Collisions, i.e. when the initial CONNECT PDU is sent simultaneously from both sides, are handled by the protocol and the connection restarted after a randomized back of timeout.

5.1.2.2 Connect Protocol Description

Unless a Connection Manager has been configured to create a connection to a peer no messages are sent and the Connection Manager doesn't respond to connection attempts.

A-side

- When a Connection Object is created at A by calling `create_conn()` A starts from DISCONNECTED state, sends a CONNECT PDU to B and enters state CONNECTING_1.
 - a. If tmo, go back to step 1 and try again after a grace period.
 - b. If B replies with a CONNECT_ACK PDU, move to CONNECTED state, send ACK PDU to B, notify RLNH that a connection have been established, and start the Connection Supervision function.
 - c. If any other PDU is received from B send RESET and go back to step 1 and try again after a short grace period.

B-side

1. After configuration, B waits in state DISCONNECTED for a CONNECT PDU from A.
 - a. If a CONNECT PDU arrives, send a CONNECT_ACK PDU and go to state CONNECTING_2
 - b. If some other PDU arrives, send RESET to A and go back to step 2.
2. In state CONNECTING_2, B waits for an ACK PDU from A.
 - a. When CONN_ACK arrives, the connect protocol is complete, the Connection Manager notifies higher layers that a connection have been established, and start the Connection Supervision function.
 - b. If some other PDU is received or the timer fires, send RESET and go to state DISCONNECTED

Allowed messages and state changes are summarized in this state diagram. The notation [xxx/yyy] means: event xxx causes action yyy.

Figure 5.2 Connection protocol state diagram

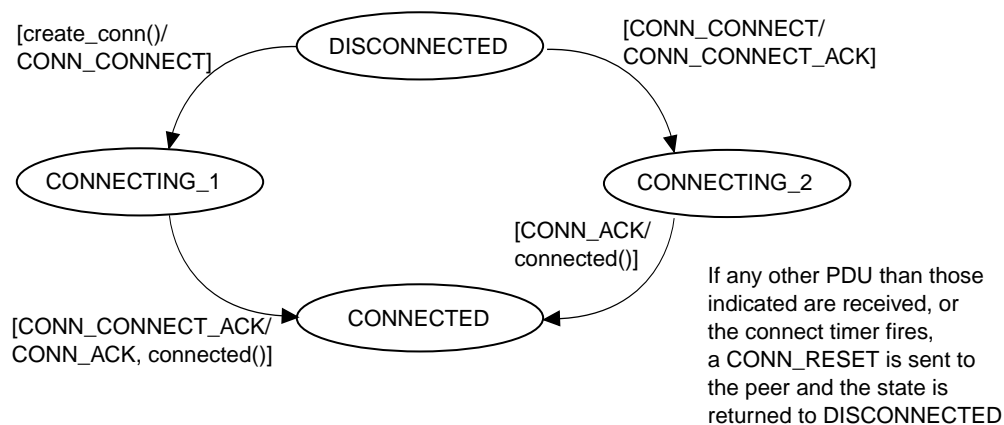
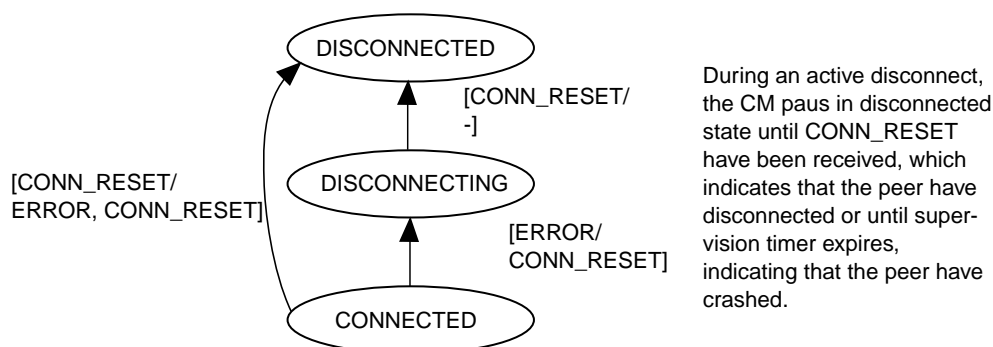


Figure 5.3 Disconnect state diagram

5.1.2.3 Feature Negotiation

A Feature Negotiation string is sent during connection establishment. The string is sent in the CONNECT ACK and the ACK type of ETHCM_CONN messages; the other messages contain an empty string '\0'). The string contains all feature names and corresponding argument. Only the features used by both peers (the intersection) are enabled. All the features supported by none or one peer (the complement) are disabled by both peers. An optional argument can be specified and It is up up to each feature how the argument is used and how the negotiation of the argument is performed.

5.1.2.4 ETHCM_CONN Header

The Connection header varies in size depending on the size of the address on the media, on Ethernet it is 16 bytes.

Table 5.3 ETHCM Connect header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				Type				Size				Window				Reserved								Conn_ID							
Dst media address followed by src media address																															
...																															
...																															
Feature Negotiation string (variable length, null-terminated string)																															

Next	Next header, the protocol number of the following Enea LINX header or 1111b if last header.
Type	CONNECT. Start a connect transaction. CONNECT ACK. Reply from passive side. ACK. Confirm that the connection have been created. RESET. Sent if any error occurs. Also sent if the next step in connect protocol fails to complete within allowed time.
Size	Media address size.

Window	Window size. Power of 2, thus Window == 5 means a window of $2^5 == 32$ packets.
Reserved	Reserved for future use, must be 0.
Conn_ID	Use this Connection ID. Informs peer which connection ID to use when sending packets over this connection. Conn ID == 0, means don't use Connection ID.
Dst and src media addresses	Dst media address immediately followed by src media address.
Feature Negotiation string	String containing feature name and argument pairs. Example: "feature1:arg1,feature2:arg2\0".

5.1.3 Enea LINX User Data Protocol

All messages originating outside the Connection Manger are sent as USER_DATA. There are two types of USER_DATA header. The first type is used for messages not requiring fragmentation and for the first fragment of fragmented messages. The second type is used for all remaining fragments.

5.1.3.1 User data and Fragmentation Protocol

A-side

1. Accept a new message from RLNH. Calculate how many fragments are required to send this message.
2. Frag_cnt = 0.
3. For each fragment in the message:
 - a. If first fragment send as USER_DATA else send as FRAG.
 - b. If only fragment set fragno to -1 else set fragno to frag_cnt and increment frag_cnt.
 - c. If last fragment set MORE to 0 else set MORE to 1.
 - d. Forward the packet to lower layer for transmission.
 - e. If last fragment go back to step 1

B-side

1. When a USER_DATA or a FRAG packet arrives.
2. If fragno = -1 (0x7fff) deliver() to RLNH since this is a complete message and wait for next packet.
3. If fragno \neq -1 find the reassembly queue for this message and add the packet to the tail of the queue (lower layers doesn't emit packets out-of-order). If the packet is the last fragment deliver the complete message to RLNH else wait for next packet.

5.1.3.2 ETHCM_UDATA Header

Table 5.4 ETHCM User Data header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				O	Reserved											M	Frag no														
Dst addr																															

Src addr																																
Next	Next header, the protocol number of the following Enea LINX header or 1111b if last header.																															
O	OOB bit, the UDATA is out of band.																															
Reserved	Reserved for future use, must be 0.																															
M	More fragment follows.																															
Frag no	Number of this fragment. Fragments are numbered 0 to (number of fragments - 1). Unfragmented messages have fragment number -1 (0x7fff).																															
Dst addr	Opaque address (to CM) identifying the receiver.																															
Srd addr	Opaque address identifying the sender.																															

5.1.3.3 ETHCM_FRAG Header

Table 5.5 ETHCM Fragment header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				Reserved												M	Frag no														

Next	Next header, the protocol number of the following Enea LINX header or 1111b if last header.																															
Reserved	Reserved for future use, must be 0.																															
M	More fragment follows.																															
Frag no	Number of this fragment. Fragments are numbered 0 to (number of fragments - 1). Unfragmented messages have fragment number -1 (0x7fff).																															

5.1.4 Enea LINX Reliability Protocol

The Connection Manager uses a selective repeat sliding window protocol with modulo m sequence numbers. The ack header carries sequence and request numbers for all reliable messages sent over the connection. Sequence and Request numbers are 12 bits wide and m is thus 4096.

In the Selective Retransmit Sliding Window algorithm the B-side explicitly request retransmission of dropped packets and remembers packets received out of order. If the sliding window is full, the A-side queues outgoing packets in a deferred queue. When space becomes available in the window (as sent packets are acknowledged by B) packets are from the front of the deferred queue and sent as usual. A strict ordering is maintained, as long as there are packets waiting in the deferred queue new packets from RLNH are deferred, even if there is space available in the window.

5.1.4.1 Reliability Protocol Description

For sliding window operation the sender A have the variables SN_{min} and SN_{max} , SN_{min} points to the first unacknowledged packet in the sliding window and SN_{max} points at the next packet to be sent. In addition the sliding window size is denoted n and size of sequence numbers are modulo m . The receiver side B maintains a variable RN which denotes the next expected sequence number. In the protocol description, the sequence

number and the request number of a packet is called **sn** and **rn** respectively. The module number **m** must be $\geq 2n$. In version 2 of the Enea LINUX Ethernet protocol **m** is 4096 and **n** is a power of 2 ≤ 128 . Drawing sequence number from a much larger space than the size of the window allows the reliability protocol to detect random packets with bad sequence numbers.

At A, the algorithm works as follows:

1. Set modulo variables SN_{min} and SN_{max} to 0.
2. In A if a message is sent from higher layer or there are packets in the defer queue, and $(SM_{max} - SN_{min}) \bmod m < n$, accept a packet into the sliding window set **sn** to SN_{max} and increment SN_{max} to $(SN_{max} + 1) \bmod m$. If $(SN_{max} - SN_{min}) \bmod m \geq n$ defer sending the packet, i.e. queue the packet until there is room in the sliding window.
3. If an error free frame is received from B containing **rn**, and $(RN - SN_{min}) \bmod m \leq (RN - SN_{max}) \bmod m$, set SN_{min} to **RN** and remove packets with **sn** $\leq SN_{min} \bmod m$ from send queue.
4. If a NACK frame is received, retransmit NACKed packets in-order with **rn** set to **RN**.
5. At arbitrary times but within bounded delay after receiving a reliable packet from B and if there are unacknowledged packets in the sliding window, send the first un-acked packet with **rn**, **RN** and the request bit set.

The selective repeat algorithm at B:

1. Set the modulo **m** variable **RN** to 0.
2. When an error free frame is received from A containing **sn** equal to **RN**, release the packet as well as following queued packets with consecutive **sn** to higher layer and increment **RN** to $(\text{last released sn} + 1) \bmod m$.
3. When an error free frame is received from A containing **sn** in the interval $RN < sn < RN + n$, put the packet in sequence number order in the receive queue and send NACK requesting retransmission of sequentially dropped packets to A. The seqno field in the NACK frame shall contain **RN** and the count field shall contain the number of missing packets.
4. At arbitrary times but within a bounded delay after receiving an error free frame from A transmit a frame containing **RN** to A. If there are frames in the receive queue send a NACK indicating missing frames.

Note that only user data is sent reliable, i.e. consume sequence numbers. ACKR, NACK and empty ACK are unreliable messages sequence numbers are not incremented as they are sent.

5.1.4.2 ETHCM_ACK header

Table 5.6 ETHCM Ack header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				R	Res				Ackno										Seqno												

Next Next header, the protocol number of the following Enea LINUX header or 1111b if last header.

R ACK-request, peer shall respond with an ACK of its own as soon as possible. (Used during connection supervision.)

Res	Reserved for future use, must be 0.
Ackno	Shall be set to the next expected or missing seqno from peer.
Seqno	Incremented for every sent packet. Note that seqnos are set based on sent packets rather than sent bytes as in TCP.

5.1.4.3 ETHCM_NACK header

NACK is sent when a hole in the stream of received reliable packets is detected. If a sequence of packets is missing all are NACKed by the same NACK packet. A timer ensures that NACKs are sent as long as there are out-of-order packets in the receive queue.

Table 5.7 ETHCM Nack header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next				Reserved				Count								Res				Seqno											

Next	Next header, the protocol number of the following Enea LINX header or 1111b if last header.
Reserved	Reserved for future use, must be 0.
Count	Number of NACKed seqnos.
Res	Reserved for future use, must be 0.
Seqno	First NACKed seqno, the rest are assumed to follow consecutively seqno + 1, seqno + 2, ... , seqno + count - 1.

5.1.5 Enea LINX Connection Supervision Protocol

The purpose of the Connection Supervision function is to detect crashes on the B-side. If B has been silent for some time A sends a few rapid pings and, if B doesn't respond, A decides that B has crashed, notifies higher layers, and initiates teardown of the connection. There is no separate PDU for the Connection Supervision function, the ACKR header from the reliability protocol doubles as ping PDU.

5.1.5.1 Connection Supervision protocol description

A-side

When a connection has been established, the Ethernet Connection Manager initializes the Connection Supervision function to state PASSIVE and starts a timer which will fire at regular interval as long as the link is up.

1. When the timer fires, the Ethernet Connection Manager checks if packets have been received from B since the last time the timer was active.
 - a. If yes, clear packets counter, set Connection Supervision state to PASSIVE and go to step 2.
 - b. If no, check if connection supervision is already in active state.
 - i. If no, enter state Active and send an ACKR to B.
 - ii. If yes, check if the ping limit has been reached.
 - A. If yes, the connection is down, notify higher layers and go to state Disconnected.
 - B. If no, resend ACKR and increment the ping counter.

2. Restart the timer.
3. If the connection is closed, stop the send timer.

B-side

- When an ACKR is received, reply to the sender with an empty ACK [RN,SNmax-1].

Figure 5.4 State diagram for Connection Supervision

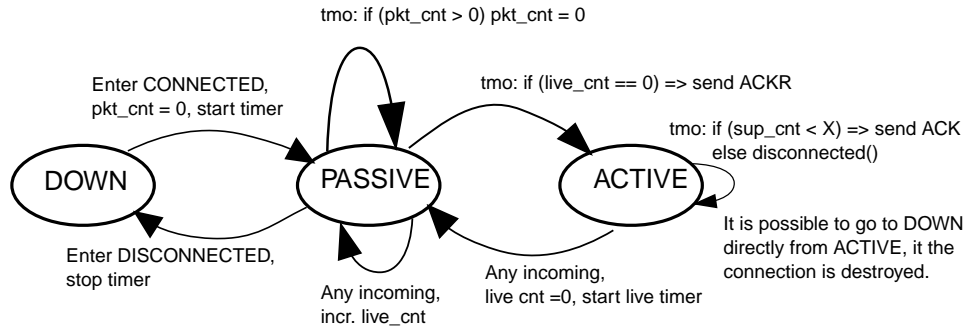
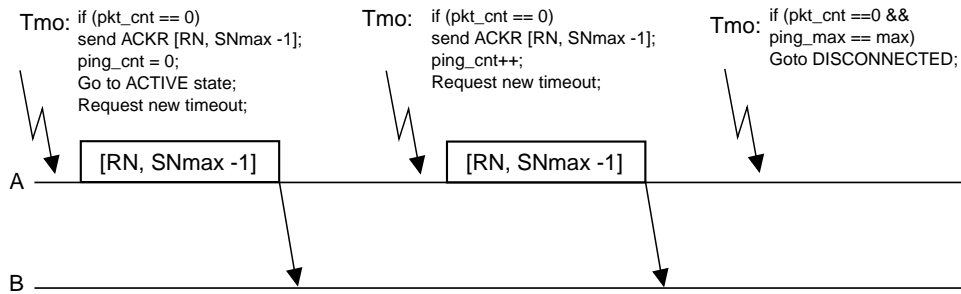


Figure 5.5 Connection supervision: the peer is down.



5.2 LINX Discovery Daemon

The LINX Discovery Daemon, **linxdisc**, automatically discovers LINX network topology and creates links to other LINX hosts. Linxdisc finds LINX by periodically broadcasting and listen for LINX advertisement messages on available Ethernet interfaces. When advertisements from other linxdiscs arrive linxdisc examines the messages and if it passes a number of controls a LINX connection is created to the sending host. When a collision occurs, i.e. two or more nodes advertising the same name, collision resolution messages are exchanged to determine which node gets to remain in the cluster. The node that wins is the one that has been up for the longest time, if the nodes can't agree which one has been up longest, the node with the highest MAC-address wins. All Linxdisc messages from this version and forward carries a version number. For this implementation of Linxdisc all messages containing a higher version number are discarded, allowing newer versions of Linxdisc to revert to version 1 of the Linxdisc protocol.

5.2.1 Linxdisc protocol

Connection procedure:

A-side

At startup read configuration file, build a list of all active interfaces, extract hostname and cluster name.

- Periodically broadcast advertisement messages on interfaces allowed by the configuration.

B-side

- When an advertisement message arrives, linxdisc performs the following steps to determine whether to connect to the sender:
 - Version number matches.
 - Not my own message, i.e. the sender address is the same as one of my interfaces.
 - Network cluster name is same as configuration.
 - Name doesn't match mine, if the name is identical to my name send a collision resolution message back to sender.
 - Name doesn't match any already established connections.
 - Check that the configuration doesn't forbid connections to this host.
- If all tests pass create a connection to the peer using the advertised name and store information about the connection for later.
- When terminated, linxdisc closes all connections it has created before exiting.
- If the configuration is changed, closes connection not allowed by the new configuration, and updates advertisements messages to reflect the new configuration.

When a collision occurs:

A-side

- Send a collision resolution message containing uptime and preferred decision.

B-side

- When receiving a collision resolution message
 - Check uptime and preferred decision if in agreement with sender refrain from sending advertisements if not in agreement, break tie based on MAC-address

5.2.1.1 Linxdisc Advertisement Message

Table 5.8 Linxdisc Advertisement Message

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Last 2 bytes Ethernet header																Version															
Type																Reserved															
Uptime sec																															

Uptime usec
Linklen
Netlen
Strings

Version	Linxdisc version type (1).
Type	Linxdisc message type, for advertisement (2) is used.
Reserved	Reserved for future use.
Uptime sec	The seconds part of the uptime.
Uptime usec	This microsecond part of the uptime.
Linklen	Length of linxname, c.f. below.
Netlen	Length of netname, c.f. below.
Strings	Carries two null-terminated strings, Linkname and Netname (network cluster name). Link name is a unique identifier for the sending host ment to be used as the name of the link created by the receiving linxdisc. Network cluster name is a unique cluster id string identifying a group of hosts that will form a LINX cluster .

5.2.1.2 Linxdisc Collision Resolution Message

Table 5.9 Linxdisc Collision Resolution Message

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Last 2 bytes Ethernet header																Version															
Type																Reserved															
Uptime sec																															
Uptime usec																															
Preferred decision																															

Version	Linxdisc version type (1).
Type	Linxdisc message type, for collision resolution (3) is used.
Reserved	Reserved for future use.
Uptime sec	The seconds part of the uptime.
Uptime usec	This microsecond part of the uptime.
Preferred decision	Preferred decision on whether the receiver should exit from the network.

6. Enea LINUX Point-To-Point (PTP) Connection Manager

This chapter describes version 1 of the Enea LINUX Point-To-Point Connection Manager Protocol.

The PTP Connection Manager is designed to meet the following requirements:

- Generic design - The design should allow several shared memory concepts. It should be easy to port the solution to other hardware. The generic parts should be separated for reuse with interfaces supporting as many different underlying layers as possible.
- Performance - The throughput should be as high as possible and the latency as low as possible. This includes minimizing the number of data copying and use of locks.
- Robust - The data exchange should be performed as simple as possible. Minimizing the complexity of the solution will minimize the possibility of making errors.

This CM is intended to communicate over reliable media, but in case of one side failure, the peer side is able to detect and react, as described in [Section 6.1.2, PTP Connection Supervision Protocol](#) [30].

6.1 Protocol Description

Enea LINUX PDUs are stacked in front of, possible, user data to form an Enea LINUX packet. All PDUs contain a "next header" field which indicates the type of the next PDU to be sent. Everything from the transmit() down call, including possible control plane signaling, from the RLNH layer is sent reliably as user data. The first field in all headers is the current type, followed by "next" field.

If a malformed packet is received the PTP Connection Manager resets the connection and informs RLNH.

When the PTP Connection Manager encounters problems which prevents delivery of a message or part of a message it must reset the connection. If the peer replies with RESET, or if the peer has crashed and the Connection Supervision timer fires, the PTP CM will notify the RLNH.

6.1.1 Enea LINUX Point-To-Point Connection Manager Headers

The Enea LINUX Point-To-Point Connection Manager protocol defines these headers.

Table 6.1 PTP Connection Manager Protocol Headers

Protocol number	Value	Definition
PTP_CM_MAIN	0x01	Main header sent first.
PTP_CM_CONN_RESET	0x02	Reset header. Used to notify the remote side that the connection will be torn down and reinitialized.
PTP_CM_CONN_CONNECT	0x03	Connect header. Used to establish connections.
PTP_CM_CONN_CONNECT_ACK	0x04	Notify remote side that connection was established.
PTP_CM_UDATA	0x05	All messages generated outside the Connection Manager are send as UDATA.
PTP_CM_FRAG	0x06	Fragment header. Messages larger than MTU are fragmented into several packages. The first fragmented package has a PTP_CM_UDATA header followed by PTP_CM_FRAG header, the following packages has only PDU PTP_CM_FRAG.
PTP_CM_HEARTBEAT	0x07	Header for heart beat control messages. These messages are used to discover a connection failure.

PTP_CM_HEARTBEAT_ACK	0x08	Acknowledgement header for heartbeat messages. Sent as a reply to PTP_CM_HEARTBEAT messages.
PTP_CM_NONE	0xFF	Indicates that the current header is the last in the PDU.

6.1.1.1 PTP_CM_MAIN Header

All messages originating outside the Connection Manager begin with MAIN header. This header specifies the type of the next header to come.

Table 6.2 PTP_CM Main header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next Header Type																															

Next Header Type Specify what type of PDU will be next.

6.1.1.2 PTP_CM_UDATA Header

All messages originating outside the Connection Manager are sent as USER_DATA. There are two types of USER_DATA header. The first type is used for messages not requiring fragmentation and for the first fragment of fragmented messages. The second type - PTP_CM_FRAG is used for all remaining fragments.

Table 6.3 PTP_CM User Data Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next Header Type								Total Size																							
Total Size								Upper layer data 1																							
Upper layer data 1								Upper layer data 2																							
Upper layer data 2																															

Next Header Type Specify what header type will follow in the current PDU.

Total Size Size of user data to be sent. If the size exceeds MTU, there message will be fragmented.

Upper layer data 1 Source address. CM does not modify this field.

Upper layer data 2 Destination address. CM does not modify this field.

6.1.1.3 PTP_CM_FRAG Head

At destination, fragments will be expected to arrive in order, but may be interrupted by other packets, and the message will be re-composed based on total size information.

Table 6.4 PTP_CM Fragment Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							

Next Header Type	Size	
Size	ID	

Next Header Type Specify what header type will follow in the current PDU.

Size The size of the fragment.

ID Fragment ID.

6.1.1.4 PTP Connection Manager Control Header

Table 6.5 PTP_CM Control Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Next Header Type								CM Version																							

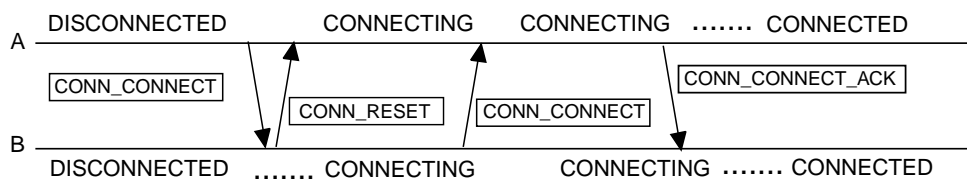
Next Header Type Type for next packet in PDU. Currently it is PTP_CM_NONE only.

CM Version Connection Manager version

6.1.1.5 PTP Connection Protocol Description

Unless a Connection Manager has been configured to create a connection to a peer, no messages are sent and the Connection Manager does not respond to connection attempts. Below is described one case when a connection is created and B side initiates the connection later then A side.

Figure 6.1 Connect Sequence diagram for PTP Connection Manager



A-side

- When a Connection Object is created, A moves from DISCONNECTED state to CONNECTING, then sends a PTP_CM_CONN_CONNECT to B.
 - a. If B replies with a PTP_CM_CONN_RESET PDU, A remains in CONNECTING state.
 - b. If B sends a PTP_CM_CONN_CONNECT PDU, A moves to CONNECTED state, sends PTP_CM_CONN_CONNECT_ACK PDU to B and notifies RLNH that a connection have been established.
 - c. If PTP_CM_CONN_RESET or PTP_CM_CONN_CONNECT PDUs are received, while in CONNECTED state, A side returns to disconnected state.

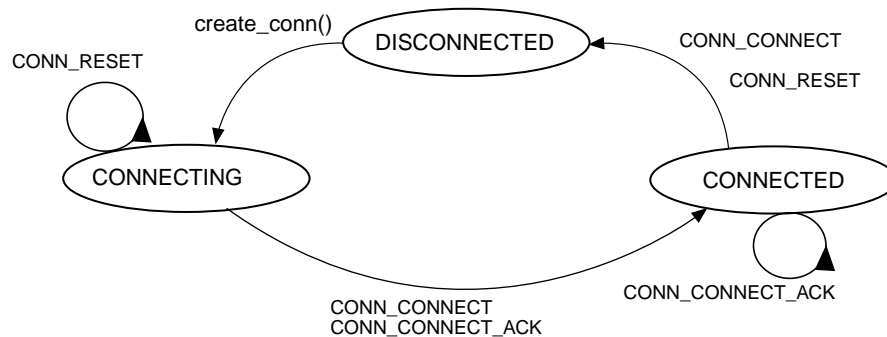
B-side

1. In DISCONNECTED state, B does not evolve from this state unless a local attempt to create a connection is made.

2. When a Connection Object is created, B moves from DISCONNECTED state to CONNECTING and sends a PTP_CM_CONN_CONNECT to A.
 - a. When PTP_CM_CONN_CONNECT arrives, A is waiting already in CONNECTING state so it enters CONNECTED state and responds to B with PTP_CM_CONN_CONNECT_ACK.
 - b. When receiving PTP_CM_CONN_CONNECT_ACK, B side enters in CONNECTED state too.
 - c. If PTP_CM_CONN_RESET or PTP_CM_CONN_CONNECT PDUs are received while in CONNECTED state, B side returns to disconnected state

If either sides receive PTP_CM_CONN_CONNECT message while in CONNECTED state, it moves to DISCONNECTED and sends a PTP_CM_CONN_RESET message.

Figure 6.2 State diagram for PTP Connection Manager



6.1.2 PTP Connection Supervision Protocol

The purpose of Supervision function is to detect crashed on peer side. In order to do that, PTP_CM_HEARTBEAT PDU is sent periodically to the other side. When a PTP_CM_HEARTBEAT_ACK is received, an internal counter is re-set. If no acknowledgement is received within a certain time interval, the heartbeat counter is decremented. After a number of ACK missed, the connection is considered crashed and will be disconnected. The timeout and the number of acknowledgments are configurable.

The supervision activity is independent of user data flow over the connection. Each side is responsible to detect peer's activity, and each side will request acknowledgment to own heartbeat messages.

7. Enea LINUX TCP Connection Manager

This chapter describes version 3 of the Enea LINUX TCP Connection Manager Protocol.

The TCP Connection Manager uses a TCP socket (SOCK_STREAM) as a connection between two LINUX endpoints. As the TCP protocol is reliable, the CM itself has no mechanism for reliability of its own. This CM is suitable to use across the internet.

7.1 TCP CM Protocol Descriptions

With the Enea LINUX TCP Connection Manager Protocol, a connection is established in the following manner.

- The TCP CM listens on port 19790 by default. Node A wants to connect to node B. A creates a TCP socket and connects it to B, sends a TCP_CONN message and then waits for a randomly amount of time for an acknowledgement. If an acknowledgment is not received, A will restart the connection procedure.
- B accepts the socket and when B wants to connect to A, it will lookup the previously accepted socket and read the TCP_CONN header. Then, it will send an acknowledgement TCP_CONN header to A.
- B considers the connection established if the send was successful and then notifies the upper layer of the established connection.
- A receives the TCP_CONN header and notifies the upper layer of the connection.

If both nodes try to connect to each other at the same time, neither of the nodes will receive an acknowledgment since the headers are sent on different sockets. This will lead to retries of the connection procedure. The timeouts for the retries are random.

7.1.1 TCP Connection Manager Headers

The Enea LINUX TCP Connection Manager protocol defines the following header and package types.

Table 7.1 TCP Connection Manager Protocol Header Types

Protocol number	Value	Definition
TCP_CONN	0x43	Connect type. Used for connection acknowledgement
TCP_UDATA	0x55	User data type
TCP_PING	0x50	Keep-alive header type
TCP_PONG	0x51	Keep-alive response header type

7.1.1.1 TCP CM Generic Header

All messages in the TCP CM protocol have the following header. Only if Type indicates TCP_UDATA the fields source and destination are used - otherwise they must be set to zero. The size field is always used in the TCP_UDATA header and it may also be used in the TCP_CONN header.

Table 7.2 TCPCM Generic Header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Reserved															O	Version								Type							

Source	
Destination	
Size	

Reserved	Reserved for future use, must be 0.
O	OOB bit, the TCP_UDATA is out of band.
Version	Version of the TCP CM protocol.
Type	Type of the current packet
Source	Source link id. Used in type TCP_UDATA, otherwise 0.
Destination	Destination link id. Used in type TCP_UDATA, otherwise 0.
Size	Size of user data in bytes, followed by the header. Used in type TCP_UDATA and TCP_CONN, otherwise 0.

7.1.1.2 TCP CM TCP_UDATA Header

All messages that don't originate from the Connection Manager are sent as TCP_UDATA. The user data is preceded by the TCP CM header with type TCP_UDATA.

7.1.2 TCP CM Connection Supervision Protocol

The two endpoints of a connection send TCP_PING headers to one another every configurable amount of milliseconds (default is 1000). When an endpoint receives a TCP_PING header, it will respond by sending a TCP_PONG header to the peer. If the connection goes down in any way, this will be detected and the CM will report this to the upper layer.

8. Enea LINX Gateway Protocol

This chapter describes the Enea LINX Gateway protocol.

The Gateway consists of a client part and a server part. The communication channel from the client to the server is a TCP connection. The client sends a request to the server that interprets the request and returns a reply back to the client. The client must not issue another request until it has got the reply for the previous one, there is one exception to this rule that will be described later.

8.1 Gateway Protocol Description

The requests and replies must be coded in big-endian format. All requests and replies start with a 8 byte header followed by a variable part. The content of the variable part depends on the request/reply. These request/reply pairs are described in detail below.

Table 8.1 Gateway request/reply codes (i.e. payload type)

Request/Reply	Value	Definition
InterfaceRequest	1	Retrieve the server's capabilities.
InterfaceReply	2	Return the server capabilities.
LoginRequest	3	Not used.
ChallengeResponse	4	Not used.
ChallengeReply	5	Not used.
LoginReply	6	Not used.
CreateRequest	7	Request the server to create a client instance, i.e. start a gateway session.
CreateReply	8	Client instance has been created.
DestroyRequest	9	Request the server to destroy a "client" instance, i.e. terminate a gateway session.
DestroyReply	10	Client instance has been destroyed.
SendRequest	11	Request the server to execute a <code>send</code> or <code>send_w_s</code> call.
SendReply	12	Return the <code>send/send_w_s</code> result to the client.
ReceiveRequest	13	Request the server to execute a <code>receive</code> or <code>receive_w_tmo</code> call.
ReceiveReply	14	Return the <code>receive/receive_w_tmo</code> result to the client.
HuntRequest	15	Request the server to execute a <code>hunt</code> call.
HuntReply	16	Return the <code>hunt</code> result to the client.
AttachRequest	17	Request the server to execute a <code>attach</code> call.
AttachReply	18	Return the <code>attach</code> result to the client.
DetachRequest	19	Request the server to execute a <code>detach</code> call.
DetachReply	20	Return the <code>deatch</code> result to the client.
NameRequest	21	Retrieve the server's name.
NameReply	22	Return server name.

8.1.1 Generic Request/Reply Header

All requests/replies starts with this header.

Table 8.2 Generic gateway request/reply header description

byte0	byte1	byte2	byte3	Description
payload_type				Type of request/reply, see table Table 8.1, Gateway request/reply codes (i.e. payload type) [33].
payload_len				Number of bytes for the type specific part, see the request/reply tables below.

8.1.2 Interface Request/Reply Payload

This request has two purposes. The client sends this request to retrieve information about the gateway server, e.g. supported requests, protocol version etc. It is also used as a "ping-message" to check that the server is alive, see receive request section for more information.

Table 8.3 Interface request payload description

byte0	byte1	byte2	byte3	Description
cli_version				The client implements this protocol version (100).
cli_flags				Bit field. Bit0 indicates client's endian (0=big, 1=little). Other bits are reserved.

Table 8.4 Interface reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, zero is returned, on error, -1 is returned.
srv_version				The server implements this protocol version (100).
srv_flags				Bit field. Bit0 indicates the server's endian (0=big, 1=little). Other bits are reserved.
types_len				Length of payload_types array (i.e. number of supported requests).
payload_types				Array of the supported requests. Each entry is 4 bytes.

8.1.3 Create Request/Reply Payload

This request is used to create a "client" instance on the server that the client communicates with.

Table 8.5 Interface request payload description

byte0	byte1	byte2	byte3	Description
user				Must be 0
my_name				"A client identifier". 0-terminated string.

Table 8.6 Interface reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.
pid				A handle that should be used in the destroy request.
max_sigsize				Maximum signal size that the server can handle.

8.1.4 Destroy Request/Reply Payload

This request is used to remove a "client" instance on the server, i.e. end the session that was started with the create request.

Table 8.7 Destroy request payload description

byte0	byte1	byte2	byte3	Description
pid				Destroy this client (see create request).

Table 8.8 Destroy reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.

8.1.5 Send Request/Reply Payload

This request is used to ask the gateway server to execute a `send` or `send_w_s` call.

Table 8.9 Send request payload description

byte0	byte1	byte2	byte3	Description
from_pid				Send signal with this pid as sender (<code>send_w_s</code>) or 0 (<code>send</code>).
dest_pid				Send signal to this pid.
sig_len				Signal size (including signal number).
sig_no				Signal number.
sig_data				Signal data (except signal number).

Table 8.10 Send reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.

8.1.6 Receive Request/Reply Payload

This request is used to ask the server to execute a `receive` or a `receive_w_tmo` call. It differs from other requests, because the client may send a second receive request or an interface request before it has received the reply from the previous receive request. The client may send a second receive request to cancel the first one. Beware that server may already have sent a receive reply before the "cancel request" was received, in this case the client must also wait for the "cancel reply". The client may send an interface request to the server, which returns an interface reply. This is used by the client to detect if the server has die while waiting for a receive reply.

Table 8.11 Receive request payload description

byte0	byte1	byte2	byte3	Description
timeout				Receive timeout in milli-seconds (<code>receive_w_tmo</code>) or -1 for infinity (<code>receive</code>).
sigsel_len				Number of elements in <code>sigsel_list</code> array. 0 means cancel previous receive request.
sigsel_list				Array of signal numbers to receive.

Table 8.12 Receive reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.
sender_pid				Received signal's sender (return value from sender).
addressee_pid				Received signal's original addressee (return value from addressee).
sig_len				Received signal's size (including signal number). 0 means "cancel receive request"-reply.
sig_no				Received signal number.
sig_data				Received signal's data (except signal number).

8.1.7 Hunt Request/Reply Payload

This request is used to ask the gateway server to execute a hunt call.

Table 8.13 Hunt request payload description

byte0	byte1	byte2	byte3	Description
user				Must be 0.
name_index				Hunt name, offset (in bytes) into data.
sig_index				Hunt signal data (except signal number), offset (in bytes) into data.
sig_len				Hunt signal size (including signal number), 0 if no hunt signal is supplied.
sig_no				Hunt signal number.
data				Signal and process name storage.

Table 8.14 Hunt reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.
pid				The pid returned by the hunt call or 0 if no process was found.

8.1.8 Attach Request/Reply Payload

This request is used to ask the gateway server to execute an attach call.

Table 8.15 Attach request payload description

byte0	byte1	byte2	byte3	Description
pid				Attach to this pid.
sig_len				Attach signal size (including signal number), 0 if no attach signal is supplied.
sig_no				Attach signal number.
sig_data				Attach signal data (except signal number).

Table 8.16 Attach reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.
attref				Return value from the attach call.

8.1.9 Detach Request/Reply Payload

This request is used to ask the gateway server to execute a detach call.

Table 8.17 Detach request payload description

byte0	byte1	byte2	byte3	Description
attref				Cancel this attach. Value returned in a previous attach reply.

Table 8.18 Detach reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.

8.1.10 Name Request/Reply Payload

This request is used to retrieve the gateway server's name.

Table 8.19 Name request payload description

byte0	byte1	byte2	byte3	Description
reserved				Reserved

Table 8.20 Name reply payload description

byte0	byte1	byte2	byte3	Description
status				On success, 0 is returned, on error, -1 is returned.
name_len				Length of server name, including '\0'.
name				Server name. 0-terminate string.

9. Enea LINUX Shared Memory Connection Manager

This chapter describes version 1 of the Enea LINUX shared-memory connection manager protocol.

9.1 Protocol Description

All SHMCM packets starts with a main header, followed by either a connection header or a user data header. Connection packets are used to setup, supervise and tear-down a connection. User data packets are used to transfer data from protocol layers above the connection manager. All protocol headers must be sent in network byte order.

9.1.1 Shared Memory Protocol Headers

9.1.1.1 SHMCM Main Header

All packets sent from the shared memory connection manger (SHMCM) starts with MAIN header.

Table 9.1 SHMCM main header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Type																															
Size																															

Type Packet type. Protocol version 1 only supports two types of packets, CON_PKT (1) and UDATA_1_PKT (2).

Size Packet size. Number of bytes that the packet consist of, including the main header.

9.1.1.2 CON_PKT Header

Connection packets or CON_PKTs are used to setup, supervise and tear down a connection.

Table 9.2 SHMCM CON_PKT header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Type																															
Cno																Spare															

Type Connection packet type. Protocol version 1 defines four different packets, CON_REQ (1), CON_ACK (2), CON_RST (3) and CON_ALV (4).

Cno Connection generation number. See [Connection Generation Number \[40\]](#).

Spare Not used, should be set to zero.

9.1.1.3 UDATA_1_PKT Header

User data packets or UDATA_1_PKTs are used to send data from the upper layers over the connection. The "user data" is located immediately after the UDATA_1_PKT header.

Table 9.3 SHMCM UDATA_1_PKT header

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0								1								2								3							
Cno																Msgid															
Src																															
Dst																															
Size																															
Address MSB																															
Address LSB																															

Cno Connection generation number. It is used to detect and discard "old" user data.

Msgid Message id. It is used to keep fragmented user data together.

Src Source address. CM does not modify this field.

Dst Destination address. CM does not modify this field.

Size User data size. CM does not modify this field.

Address MSB Not used in protocol version 1.

Address LSB Not used in protocol version 1.

9.1.2 SHM Connection Protocol Description

The SHMCM consider shared memory to be reliable, i.e. packets will not be lost and they are always received in the same order that they were sent by the peer. The protocol design requires that these two characteristics are fulfilled.

The protocol uses four types of connection packets, listed below, to establish, supervise and tear down a connection.

Table 9.4 SHMCM CON_PKT types

Protocol number	Value	Definition
CON_REQ	1	This packet is sent to start connection setup.
CON_ACK	2	This packet is sent as a reply to a CON_REQ.
CON_RST	3	This packet is sent to the peer to abort the connection setup or to gracefully tear down the connection. It is not allowed to send user data after that a CON_RST has been sent.
CON_ALV	4	Once the connection is setup, this packet is sent periodically to indicate that "I'm alive".

Connection Setup

When the SHMCM receives the `connect` down call from RLNH, it sends a `CON_REQ` to the peer and waits for a `CON_ACK` from the peer. When the SHMCM receives the `CON_ACK`, it calls the `connected` up call to notify RLNH that the connection is up. If the SHMCM receives a `CON_REQ` instead of a `CON_ACK`, it replies with a `CON_ACK` and calls `connected`.

Connection Established

If the SHMCM receives a `CON_ACK` at this point, it should ignore the `CON_ACK`. However, if the SHMCM receives a `CON_RST` or a `CON_REQ`, it should call the `disconnected` up call to notify RLNH that the connection is down. In case that a `CON_REQ` was received, the SHMCM must send a `CON_RST` before calling `disconnected`.

When the connection has been established, it should send `CON_ALV` periodically to tell the peer that "I'm alive". If the SHMCM has not received any packets, `CON_ALV` or user data, from the peer for a defined period of time, it should consider the peer dead and disconnect the connection, i.e. send a `CON_RST` and call `disconnected`.

If the SHMCM receives the `disconnect` down call from RLNH, it should send a `CON_RST` and call `disconnected` up call. Also, if the SHMCM encounters an internal error, it should send a `CON_RST` (if possible) and call `disconnected`.

Connection Generation Number

The SHMCM should have a 16-bit counter that is incremented before a `CON_REQ` is sent. This number should be included in all packets. The peer should store the counter value, that is received in a `CON_REQ/CON_ACK`, and compare it with the one that is included in the packets. If they differ, the packet should be discarded.

User data

The SHMCM gets the user data and corresponding meta-data in the `transmit` down call. The user data is packed into one or more `UDATA_1_PKT`s and sent to the peer. If more than one packet is required, an ID tag (`msgid`) should be added to the packets so that the peer can identify which fragments that belongs together and re-assemble them.

The SHMCM should use the size fields in the main header and the user data header to determine if the user data has been split into several `UDATA_1_PKT`s.

10. Enea LINX RapidIO Connection Manager

This chapter describes version 2 of the Enea LINX RapidIO Connection Manager Protocol. It is not compatible with version 1, which is not documented here since it has never been implemented for LINX for Linux.

10.1 Protocol Description

RapidIO is considered as a reliable media, so there is no need for a reliability protocol. RapidIO hardware are however not reliable to the full extent - packets can be received in a different order than they were sent. This requires sequence numbers on all data packets that are sent with the protocol described herein.

If a malformed packet is received, the RapidIO Connection Manager resets the connection and informs the RLNH.

When the RapidIO Connection Manager encounters problems which prevents delivery of a message or part of a message it must reset the connection. The RapidIO CM notifies the RLNH when the peer replies with RESET or, if the peer has crashed, the Connection Supervision timer fires.

All Enea LINX RapidIO packets starts with the common `rio_frame` header. The `rio_frame` header allows for other protocols to coexist with this protocol.

Table 10.1 RIOCM `rio_frame` header

0	1	2	3
protocol		size	

The protocol number is 0x22cf (8911) for version 2 of the Enea LINX RapidIO Connection Manager Protocol. It was 0x8911 for version 1.

The `rio_frame` header is followed by either a CONN, HEARTBEAT or UDATA header. There are three types of CONN headers, one HEARTBEAT header and five types of UDATA headers. Some fields are common for all nine headers. The common fields are type and `dst_port`:

Table 10.2 RIOCM common fields

0	1	2	3
type	reserved		
<code>dst_port</code>		reserved	

The type is used to identify the type of the packet. The `dst_port` is used to identify the destination port of the packet. In multi-core processors that share the same RapidIO device, the `dst_port` can be used by the hardware to identify the destination core of the packet.

10.1.1 LINX RapidIO Connection Establishment Algorithm

The following headers types are used in the Connection Establishment Algorithm:

Table 10.3 RapidIO Connection Manager Connect Headers

Header type	Value	Definition
RIO_CONN_REQ	0x81	Connect request. Used to request connection establishment and its settings

RIO_CONN_ACK	0x82	Connect acknowledgement. Used to acknowledge the connection request and its settings.
RIO_CONN_RESET	0x83	Reset. Used to cancel the connection.

The connection algorithm is quite straightforward.

Let X and Y be two nodes in the same rapidio network. Let a connection be created on X towards Y. X will start to periodically send RIO_CONN_REQ to Y. When a connection is created on Y towards X, it will do the same.

When Y (or X) receives a RIO_CONN_REQ, it will respond with a RIO_CONN_ACK. When X receives a correct RIO_CONN_ACK, it will consider the connection as established and respond with its RIO_CONN_ACK. When Y receives this RIO_CONN_ACK, it will consider the connection as established, it may also send a RIO_CONN_ACK to X again, which will be dropped by X.

The last transmission of a RIO_CONN_ACK will seem unnecessary, but it is a consequence of the state machine used in the implementation. The important thing to notice here is that a correct RIO_CONN_ACK is dropped by a peer that considers the connection to be established.

A correct RIO_CONN_ACK is a RIO_CONN_ACK with the correct negotiated settings for the connection. The RIO_CONN_REQ has fields with requests for the mtu and the heartbeat timeout for the connection. These values are set by a logic expression for each field. The result of this calculation is sent in the RIO_CONN_ACK. The lowest configured mtu is used, while the largest configured heartbeat timeout is used. If this calculation mismatches, the RIO_CONN_ACK is regarded as incorrect, and then treated as a RIO_CONN_REQ instead!

10.1.1.1 RIO_CONN_REQ Header

Table 10.4 RIO_CONN_REQ Header

0	1	2	3
type	generation	mtu	
dst_port		rsvd	hb_tmo
sender		src_port	

type	RIO_CONN_REQ
generation	A generation value for this connection. Used to distinguish between new and old connections.
mtu	The MTU that the sender of the RIO_CONN_REQ wants to use for this connection.
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
reserved	This field is not used in this header.
hb_tmo	The heartbeat timeout that the sender of the RIO_CONN_REQ wants to use for this connection. The value is written in hundreds of msec. A value of 5 yields a tmo of 500ms.
sender	The device id of the sender.
src_port	The port of the sender.

10.1.1.2 RIO_CONN_ACK Header**Table 10.5 RIO_CONN_ACK Header**

0	1	2	3
type	generation	mtu_ack	
dst_port		generation_ack	hb_tmo_ack
sender		src_port	
my_cid			

type	RIO_CONN_ADD
generation	A generation value for this connection. Used to distinguish between new and old connections. The connection generation value of the sender.
mtu_ack	The MTU calculated by the sending peer.
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
generation_ack	This field is used to acknowledge the generation field sent from the peer in the RIO_CONN_REQ.
hb_tmo_ack	The heartbeat timeout calculated by the sending peer. The value is written in hundreds of msec. A value of 5 yields a tmo of 500ms.
sender	The device id of the sender.
src_port	The port of the sender.
my_cid	The cid (connection id) that is to be used in udata and heartbeat headers for this connection. It is unique for each connection and allows the receiver to quickly identify the connection.

10.1.1.3 RIO_CONN_RESET Header**Table 10.6 RIO_CONN_RESET Header**

0	1	2	3
type	generation	mtu	
dst_port		rsvd	
sender		src_port	

type	RIO_CONN_RESET
generation	A generation value for this connection. Used to distinguish between new and old connections.
mtu	This may contain the MTU of the connection. Disregarded field.

dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
rsvd	This field is not used in this header.
sender	The device id of the sender.
src_port	The port of the sender.

10.1.2 Enea LINX RapidIO User Data Protocol

Data is sent reliable on RapidIO. But some drivers may not guarantee that the order of the incoming packets is the same as the order they were sent in. Therefore, the user data packets need sequence numbers. The receiver implements a reordering queue to ensure that all messages are delivered in order.

Table 10.7 RapidIO Connection Manager User Data Headers

Header type	Value	Definition
RIO_SINGLE	0x1	Used when the entire messages plus header can fit within the MTU
RIO_FRAG_START	0x2	The first fragment when sending with simple fragmentation
RIO_FRAG	0x3	Remaining fragments for both types of fragmentation
RIO_PATCH_START	0x4	The first fragment when receiving with improved fragmentation
RIO_PATCH	0x5	Contains patches for the 'holes' created by the improved fragmentation

Messages can be smaller or larger than the max amount of payload that can be sent in a single packet. When the messages are smaller, the RIO_SINGLE header is used. If otherwise, there are two ways of sending fragmented messages.

The simple fragmentation is implemented using two headers, RIO_FRAG_START and RIO_FRAG. A RIO_FRAG_START header is trailed by the needed amount of RIO_FRAG headers to complete the message.

The other way of sending/receiving fragmented messages is an optimization done in the OSEck Operating System. The Linux implementation of the RapidIO Connection Manager only has support of receiving such messages. This way is implemented using three headers, RIO_PATCH_START, RIO_FRAG and RIO_PATCH. The RIO_PATCH_START is trailed by the necessary amount of RIO_FRAG headers and then finally trailed by the necessary amount of RIO_PATCH headers. The scenario for using these headers is as follows:

The cm needs to send a message larger than mtu over the connection. To avoid a lot of memcpy's, the cm merely overwrites some data of the message with RIO_PATCH_START and RIO_FRAG headers after moving that data to RIO_PATCH packets. It will then transmit every fragment of the message, which then consists of subsequent data packets. Finally, it transmits all RIO_PATCH packets that the peer needs to patch (repair) the overwritten data in the message with.

10.1.2.1 RIO_SINGLE Header

Table 10.8 RIO_SINGLE Header

0	1	2	3
type	msgid	seqno	
dst_port		dst_cid	

sender	src_port
src	
dst	
payl_size	

type	RIO_SINGLE
msgid	A message identifier for this message.
seqno	The sequence number for this packet
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	The device id of the sender.
src_port	The port of the sender.
src	The source link address of the message
dst	The destination link address of the message.
payl_size	The total size of the message.

10.1.2.2 RIO_FRAG_START Header

Table 10.9 RIO_FRAG_START Header

0	1	2	3
type	msgid	seqno	
dst_port		dst_cid	
sender		src_port	
src			
dst			
payl_size			

type	RIO_FRAG_START
msgid	A message identifier for this message.
seqno	The sequence number for this packet
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	The device id of the sender.

src_port	The port of the sender.
src	The source link address of the message
dst	The destination link address of the message.
payl_size	The total size of the message.

10.1.2.3 RIO_FRAG Header

Table 10.10 RIO_FRAG Header

0	1	2	3
type	msgid	seqno	
dst_port		dst_cid	
sender		src_port	

type	RIO_FRAG
msgid	A message identifier for this message.
seqno	The sequence number for this packet
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device. When multiple connections share a device ID they must have different ports.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	The device id of the sender.
src_port	The port of the sender.

10.1.2.4 RIO_PATCH_START Header

Table 10.11 RIO_PATCH_START Header

0	1	2	3
type	msgid	seqno	
dst_port		dst_cid	
sender		src_port	
src			
dst			
payl_size			
count_frag		count_patch	

type	RIO_FRAG_START
msgid	A message identifier for this message.
seqno	The sequence number for this packet

dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	The device id of the sender.
src_port	The port of the sender.
src	The source link address of the message
dst	The destination link address of the message.
payl_size	The total size of the message.
count_frag	Number of RIO_FRAG fragments that will trail this header
count_patch	Number of RIO_PATCH packets that will trail the fragments

10.1.2.5 RIO_PATCH Header

Table 10.12 RIO_PATCH Header

0	1	2	3
type	msgid	seqno	
dst_port		dst_cid	
sender		src_port	

type	RIO_PATCH
msgid	A message identifier for this message.
seqno	The sequence number for this packet
dst_port	This is the user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	This is the device id of the sender.
src_port	This is the port of the sender.

10.1.3 Enea LINUX RapidIO Connection Supervision Protocol

Supervision of the connection is needed in order to detect if a peer has ‘died’.

Table 10.13 RapidIO Connection Manager Connection Supervision Header

Header type	Value	Definition
RIO_HEARTBEAT	0x6	Sent periodically to indicate that the sender is alive

The heartbeat packet is sent every negotiated $hb_tmo * 100$ ms. When a node has not received a heartbeat in three periods, the connection is considered timed out and has to be reestablished.

10.1.3.1 RIO_HEARTBEAT Header

Table 10.14 RIO_HEARTBEAT Header

0	1	2	3
type	pad	rsvd	
dst_port		dst_cid	
sender		src_port	

type	RIO_HEARTBEAT
pad	pad
rsvd	Not used.
dst_port	The user defined destination port of the connection. This field is used by some RapidIO hardware to distinguish between destination cores that shares the same device.
dst_cid	The connection id that was submitted in the RIO_CONN_ACK header.
sender	The device id of the sender.
src_port	The port of the sender.

Index

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#)
[N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

C

connection manager, [7](#)
 Ethernet, [15](#)
 point-to-point, [27](#)
 RapidIO, [41](#)
 TCP, [31](#)
connection manager protocol, [14](#)

E

eupervision, [14](#)

F

feature negotiation, [9](#), [19](#)

G

gateway, [33](#)

H

heterogeneous systems, [7](#)
hunt, [10](#)

I

identifier, [9](#)
IPC, [7](#)

L

link address, [9](#)
LINX cluster, [26](#)
linxdisc, [24](#)

O

out of band, [21](#), [32](#)

P

point-to-point, [27](#)
PTP, [27](#)

R

reliability, [21](#)
reliable, [14](#)
reliable media, [27](#)
RLNH, [7](#), [9](#)

S

SOCK_STREAM, [31](#)
supervision, [10](#), [23](#), [30](#), [32](#)