OpenLCB SpecificationWorking Note	
OpenLCB Message Network	
Dec 31, 2014	Preliminary

This Working Note contains informative discussion and background for the OpenLCB CAN Message Network. It's under active development, which means it's messy. It is written in several voices because parts will eventually become a Standard and other parts a Technical Note.

1 Introduction

This explanatorySpecification (future Technical Note) contains informativenormative information about the OpenLCB Message Network, corresponding discussion and background for the (eventual) can be found in the corresponding "OpenLCB CAN Message Network SpecificationTechnical Note".

The protocol is described via three components: the state machine within the node(s), the messages, and the basic interactions that the nodes must take part in. These are separate described below. The messages are described in terms of a general format, and specific message definitions.

Messages are transported across a specific data-link level implementation, for example using CAN frames or TCP/IP sockets. The messages are described first in general terms, then mapped to specific implementations. The states and interactions are the same across all data-link implementations.

Annotations to the Specification

This section provides background information on corresponding sections of the Specification document. It's expected that two documents will be read together.

Introduction

OpenLCB is based on a global exchange of individual messages. This specification defines the basic messages and how they interact. Higher-level protocols are based on this message network, but are defined elsewhere.

This specification separately describes how messages and/or parts of messages are transported across CAN segments within OpenLCB-CAN format frames.

This specification separately describes how messages and/or parts of messages are transported across TCP/IP sockets.

2 Intended Use

The interactions described here are used by all OpenLCB nodes to connect to the OpenLCB network.

~~All OpenLCB protocols are built upon the exchange of messages. Higher-level protocols for exchange of data via datagrams, events and for other purposes are constructed using messages as described here.~~

2.1 References and Context

For background information on format and presentation, see:

- OpenLCB Common Information Technical Note

This specification is in the context of the following OpenLCB Specifications:

- The OpenLCB Node Identifier Specification, which specifies Node Identifiers and how they are defined.

This specification is in the context of the following OpenLCB-CAN Specifications:

- The OpenLCB-CAN Frame Transfer Specification, which specifies transfer of OpenLCB messages over CAN segments. “CAN” refers to the electrical and protocol specifications as defined in ISO 11898-1:2003 and ISO 11898-2:2003 and their successors.

This specification is in the context of the following OpenLCB-TCP/IP Specifications:

- The OpenLCB-TCP/IP Segment Transfer Specification, which specifies transfer of OpenLCB messages over TCP/IP links.

Conformance with a later version of a referenced standard shall be accepted as conformance with the referenced versions.

2.2 Message Format

OpenLCB messages are sent using the transfer mechanism and format described in the specification for a specific wire protocol.

All messages shall contain a source Node ID and a Message Type Indicator (MTI). The MTI defines both the general format of the message and its specific type. All messages with the same MTI are of the same type.

2.2.1 Message Type Indicators

(Most of this section should be moved to the end as a section 3, past the Standard annotation)

The general Message Type Indicator (MTI) is a 16-bit quantity, although it may be remapped for specific wire protocols.

Many nodes will treat MTIs as magic 16-bit numbers, just comparing them for equality to specific values of interest. That's a perfectly fine node implementation strategy.

This section describes how the numeric values for those MTIs are allocated. The current allocations are documented in a separate spreadsheet[1]. We keep them in just that one place to avoid conflicting updates. Those allocations are normative. The discussion in this section is not normative on OpenLCB users or node developers, but does describe the methods that are to be used for allocating new MTI values for new OpenLCB message and protocol types.

(This is just informative, not normative; it's the actual MTI values that are normative, not how they were picked; what we're trying to do here is resolve documentation conflicts in advance, and it's not really working)

Because the MTI values are specified for each kind of message, the Standard just documents those results. This section of the Technical Note addresses the method for choosing specific values.

We've chosen to allocate MTI bit fields to make decoding simpler; if possible, aligned on nibble boundaries to make it easy to read as hexadecimal numbers. We've also used a mix of bit-fields and individual flag bits to increase compatibility when additional MTI values are defined later.

There are two basic approaches to identify classes of message types, such as “addressed” vs “global” messages.

1. Use a dedicated bit field to distinguish the types, e.g. 1 indicates addressed and 0 means global.
2. Encode in the type number, e.g. “Type A (addressed) is 1”, “Type B (addressed) is 2”, “Type C (global) is 3”, “Type D (addressed) is 4”.

The encoded form uses less bits, particularly if there are many classes to distinguish, which would require many dedicated bits. But future expansion is easier with dedicated bit fields, because nodes can do some limited decoding of MTIs even though the node was created before the new MTI values were defined. For example, a gateway can determine whether a message is global or addressed to a particular node, even if the specific MTI was defined after the node was built.

Field Name	Size & Location mask	Description
Reserved	2 bits 0xC000	Send and check as zero.
Special flag	1 bit 0x2000	0 means original-segment only, 1 should be forwarded throughout. 0 is used for segment-specific commands.
Stream or Datagram	1 bit 0x1000	1 means stream or datagram, 0 means regular message.
Static Priority	2 bits	0 to 3

[1]See http://openlcb.org/specs/index.html for the current version of the spreadsheet. It provides concrete examples that may help you understand the material in this document.

	0x0C00	0 goes first, 3 last if priority processing is present
Type Within Priority	5 bits 0x03E0	
Simple Node flag	1 bit 0x0010	0 means for simple node, 1 means not
Address Present	1 bit 0x0008	
Event ID flag	1 bit 0x0004	1 means Event ID present, 0 not present
Modifier bits	2 bits 0x0003	Used in some MTIs; by default 0b00 sent and checked

Table 1: Common MTI Layout

The "Special" bit defines whether a message on a segment, e.g. a TCP/IP backbone link between CAN gateways, should be propagated off that segment to others. A 0 means that the message should not go through gateways; a 1 indicates that it should.

The next bit, the "Stream or Datagram" bit, is set to indicate that this MTI is the data part of the stream or datagram protocols (defined elsewhere). For efficiency sake, these are handled in a special way on CAN and perhaps other wire protocols; this bit makes it easy to identify these MTIs.

The contents of the top nibble is still under development. Note that there are only a few values currently defined as useful: 0 and 1 for messages that will route to any CAN network; and 2 for messages that are not propagated to CAN networks. In the future, we may want to change this nibble from bit coding to value coding but those specific values will be preserved. Values 3-15 or the top 2 bits are not currently used and available for the future.

The next eight bits form a specific message type number. This has substructure:

- The most-significant two bits are used to form static priority groups. A 0 bit is considered to have more priority (can be processed first), a 1 bit less priority (can be processed later). The MSB makes a larger statement about priority than the LSB of these two. Priority processing is permitted but not required. This is included to allow protocol designers to ensure that CAN frame reordering (which, although not always present, is a normal part of CAN that must be considered) won't result in problems for communications.

  The priority mechanism may or may not work outside the priority field; MTIs should be chosen to work in either case.

- The next 5 bits, forming the low nibble, are used to indicate the specific message type within a priority group and taking into account the values of all the other bits. This is the unique part that's selected during the process for designing a new OpenLCB protocol to ensure a unique MTI.

- The least significant bit “Simple Node flag” is used to indicate messages meant for “simple” or minimal nodes. A 1 in this bit means that these simple nodes can ignore this message. A 0 in this bit means that the simple nodes must process the message. See the section ??? below for more information. This bit is reserved to 0 for all addressed message types, as a message specifically delivered to a node should be processed.

The bottom nibble of the MTI is interpreted as flags that define the structure and format of the message type.

- The most significant bit, the “Address Present” flag, indicates whether the message carries a destination address (1) or does not carry a destination address (0). These are also referred to as “addressed” or “global” messages respectively.

  Global messages shall be delivered to all nodes.[2]

  Addressed messages shall be delivered to the node in their destination node ID. They may, but need not, be delivered to other nodes.

- The next (second) bit indicates this message carries a P/C Event ID field when set to 1. Setting 0 means that this message does not carry a P/C event ID. If a P/C Event ID is present, it is at a specific location in the message content, right after the destination address (if present) or right after the MTI (if no destination ID present).

- The two least-significant bits can be used as modifiers to the specific MTI, or (later) used to create additional MTI codes. At present, unless used as MTI modifiers, these should be sent and checked as all 1 bits, 0b11. Some MTIs have additional status bits defined as part of this field. For example, there are two status bits associated with “Consumer Identified” which must be kept in the header since there is no room in the CAN data field. These can be considered as MTI modifier bits.

This MTI organization allows nodes to do simple decoding of messages with MTIs that they don't recognize, perhaps because they were defined after the node was created. For example, gateways can use this to control routing of messages that they don't understand, perhaps because they were defined after the gateway was developed.

The 0x0000 and 0x0001 MTI's are explicitly reserved. 0x0001 may be used in the future as part of a CAN hold-off mechanism; it's a very high priority which nodes can send to delay other frames. The all-zero MTI is inevitably needed for internal flags, empty buffers, etc in software implementations.

2.2.1.1 CAN MTI Considerations

MTI information is carried in a different format on CAN links to increase bandwidth efficiency, simplify decoding in small processors, and permit use of hardware filtering.

[2]The “simple node protocol” is an exception to this, which needs to be worked into this Standard.

The standard CAN MTI field is 12 bits in the header (messages without destination address) or one byte in the data segment (addressed messages). Since CAN frames only carry 8 data bytes, a 1-byte MTI short form will be used until future expansion makes more necessary. The possibility of longer MTI values has been reserved, see below.

After mapping into one or more CAN Frames, the standard MTIs are mapped to one of eight frame types:

Frame Type	Meaning
0	(Reserved)
1	Global & Addressed MTI
2	Datagram complete in frame
3	Datagram first frame
4	Datagram middle frame
5	Datagram final frame
6	(Reserved)
7	Stream Data

Table 2: MTI Type Values

The 2, 3, 4, 5 (Datagram frames) and 7 (Stream Data) values are special cases chosen for efficient processing of large amounts of data on CAN. Most MTIs will map to 0 or 6. (Reference other docs for datagram, streams)

2.2.1.1.1 Addressed and Global CAN MTIs

MTIs with the “stream or datagram” bit unset are mapped to and from type 1.

In this case, the next twelve bits of the CAN header are available for MTI information. These carry the lowest-order 12 bits of the entire MTI, plus by implication the “stream or datagram” and “special” bits as zeros.

If the “addressed” bit is set to 1, the destination address is placed in the 1st two bytes of the data part of the CAN frame. The top nibble of the 1st byte contains flags (see below); the lower nibble of the 1st byte, and the entire 2nd byte contain the 12-bit destination alias from the CAN frame level protocols.

The format of this in binary is

rrff dddd, dddd dddd

rr are two reserved bits.

The two ff bits can be used for packing and unpacking large messages to a sequence of CAN frames, see below. The coding is

- 00 Only frame
- 01 First frame of more than one
- 10 Last frame of more than one
- 11 Middle frame of more than 2.

You can think of these as active-zero start and end bits respectively.

CAN frames marked as First or Middle frame shall carry eight total data bytes. CAN frames marked as Last or Only frame shall have from two through eight total data bytes.

Field	CAN prefix	Frame Type	Static Priority	Type within Priority	Simple Node flag	Address Present	Event ID present	Modifier Bits	Source ID
Size & location (within 29-bit CAN Header)	0x1800, 0000	0x0700, 0000	0x00C0, 0000	0x003E, 0000	0x0001, 0000	0x0000, 8000	0x0000, 4000	0x0000, 3000	12 bits 0x0000, 0FFF
Value(s)	3	0	See Table 1 MTI description						12-bit alias for source node

Table 3: Unaddressed MTI CAN Header Format

2.2.1.1.2 Datagram and Stream frame format

This format is used for frames made from Datagram and Stream Data messages, as described in the specifications for those protocols, with values of 2 through 5 and 7 in the Type field respectively.

In this case, the 12-bit destination alias is placed in the header.

Since Datagram and Stream Data messages are broken into multiple frames for transmission, it's possible that errors will cause the last frame(s) to be lost. Disconnection or crashing of a node or gateway could cause this, for example. Although higher-level timeouts will protect the Datagram and Stream protocols, the message-level specification needs to cover how the frame-to-message process

recovers. The most straightforward would be a "may, but are not required to" that relies on adjacent transfer: "if more than 500 milliseconds elapses after the receipt of a first or middlr datagram frame without receipt of a middle or end frame, the node may, but is not required to, initiate error recovery by forwarding the message, marked as received in error, for further internal processing." CAN segments carry lots of traffic, so there probably needs to be something in there that allows for time lost to higher priority frames. That needs to be developed. For definiteness, the error recovery should probably be specified. It's true that a disconnected network might not transport it if the receiving node sends an error reply, but the receiving node should probably send it anyway. That allows other nodes to see what the receiver is thinking. And if it was just a transient loss, this will help keep state machines in synch. So, the node is required to send a Datagram Rejected with a specific error code, perhaps "Resend OK", "Transport Error", "Out of Order". That needs to go into the Datagram and Stream protocol specifications.

Field	CAN prefix	Frame Type	Destination ID	Source ID
Size & location (within 29-bit CAN Header)	2 bits 0x1800,0000	3 bits 0x0700,0000	12 bits 0x00FF,F000	12 bits 0x0000,0FFF
Value(s)	3	6	12-bit alias for destination node	12-bit alias for source node

Table 4: Addressed MTI CAN Header Format

2.2.1.1.3 CAN Notes

Note that the priority bit in the CAN frame is separate from the static priority field in the MTI format specification.

Because of their length, standard-header CAN frames don't carry the bits for the CAN MTI. Both hardware and software generally provide these as 0 bits, and require that the user code check an “extended” flag to know whether it's dealing with a standard or extended frame. By ensuring that CAN header part of the MTI coding can never be zero, we ensure that standard frames don't accidentally get interpreted as OpenLCB frames. (The leading priority bit's default value of 1 can't be assumed to always be present)

2.2.2 Message Content

The message content consists of:

- The source Node ID
- The MTI
- If flagged as present, the destination Node ID
- If flagged as present, an Event ID

- Any other content as defined for the specific message type

The exact format and order are defined by the specific wire protocols, but in all cases the message must be fully decodable based on the flag-bit information in the MTI.

2.3 States

The message network layer in an OpenLCB node has two states:

- Uninitialized
- Initialized

Nodes shall start in the Uninitialized state.

A node in the Uninitialized state may transmit an Initialization Complete message. A node in the Uninitialized state shall not transmit any other message type.

A node in the Initialized state may transmit any message type.

The Uninitialized state is only occupied when the node is first starting up. This makes for a really simple state machine: Just send Initialization Complete message first thing on coming up.

At present, there's no way to deliberately return to the Uninitialized state. No need for this has been identified.

2.4 Error codes

Numerous messages are defined to carry status information and error codes. An OpenLCB Error code is a 2-byte value of the following format:

Bit	Value	
Bits 12-15	Exactly one of these bits should be set.	
	0x8000	Accept, no error. This value must not be used in Reject messages.
	0x4000	Reserved. Send as 0, treat it as unspecified Permanent error.
	0x2000	Temporary error (resend OK). Re-trying the same interaction later is likely to succeed.
	0x1000	Permanent error. Re-trying the same interaction will result in the same error.
Bits 8-11	Specific error enumeration. Possible values for this field are documented in the particular protocol standard. Values may be re-used for different General error flags settings. Value of zero in this field means no further information is available.	
Bits 4-7	General error flags.	

	For Permanent error, the following flags are defined:	
	0x1080	Invalid arguments. Some of the values sent in the message fall outside of the expected range, or do not match the expectations of the receiving node.
	0x1040	The transport protocol (datagram/stream) is not supported.
	0x1020	Source not permitted.
	0x1010	Unimplemented. The functionality that the message tried to invoke is not implemented in the destination node.
	For Temporary error (resend OK), the following flags are defined:	
	0x2080	Transport error.
	0x2040	Unexpected, Out of order. An inconsistency was found in the message or frame sequence received, the arrived message is unexpected or does not match the state of the receiving node.
	0x2020	Buffer unavailable
Bits 1-3	Reserved. Send as 0, ignore on receipt.	
Bit 0	0x0001	Information logged. See the Logging protocol to retrieve the logged information. This bit may be added to any error value.

2.5 Definition of Specific Messages

This section defines the format of common core messages. Although there is a short description of the purpose of the message, this is just for identification & explanatory purposes. The meaning of the messages is defined by the interactions they appear in, which are described in later sections.

Note that Node ID in the data part of several messages is sent in full 48-bit format in all wire protocols, specifically including CAN, even if an alias or alternate form is available elsewhere in the message.

2.5.1 Initialization Complete

Indicates that the sending node initialization is complete, and once the message is delivered, reachable on the network.

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Initialization	N	N	N	0x0100	Sending Source ID

Complete					

2.5.2 Verify Node ID

Issued to determine which node(s) are present and can be reached.

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Verify Node ID	N	N	Y	0x28A7	(optional) Node ID
	Y	N		0x30A0	(optional) Node ID

There are multiple forms of the Verify Node ID message.

The global (unaddressed) format may include an optional NodeID. If present, only nodes with a matching Node ID should reply. If absent, all nodes should reply.

The addressed form may include an optional full NodeID in the data section. The addressed node must always reply, whether or not a Node ID is carried in the data, and whether there's a match when the optional Node ID is present. The optional NodeID idea just serves as documentation of the intent of the request. (Makes sense on CAN, not so much on other transports that don't use aliases).

(Should the discussion of replying be here, or under interactions? Might be better to have this section talk about the meaning of the Node ID, rather that about replies.)

(In the TN, add an example of the full format messages, which can include two copies of the Node ID due to the way destination alias mapping works)

2.5.3 Verified Node ID

Reply to the Verify Node ID message.

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Verified Node ID	N	N	Y	0x28B7	Full Node ID of sending node

The node ID in the data is redundant on wire protocols that carry the full source ID, but can be very valuable for wire protocols that abbreviate (“alias”) the source ID within the messages, e.g. CAN.

(With the new format, should there be an addressed version of this? Can always send to the source ID/alias of request, so will always work.)

This could have, we think, been an addressed message. The source information on the Verify Node ID request will be enough to get the Verified Node ID reply back to the originator. Just in case, though, we left it as a global which is defined to go everywhere, so routing of it can't fail.

2.5.4 Optional Interaction Rejected

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Optional Interaction Rejected	Y	N		0x30C0	Error codes, MTI, optional information

The contents are, in order:

- Two bytes of error code.
- Two bytes of MTI. If the frame transport only delivered part of the MTI[3], that content is returned with the rest of the MTI bits set to zero.
- Any extra bytes that the node wishes to include. There can be zero or more of these. These must be described in the node documentation.

Nodes must process this message even if not all of the contents are provided.

Error codes:

0x1000 bit: if set, this is known to be a temporary error, and the interaction can be retried.

0x2000 bit: if set, this is known to be a permanent error, and the interaction must not be retried.

2.5.5 Terminate Due to Error

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Terminate Due to Error	Y	N		0x30D0	Error code, MTI, optional information

The contents are, in order:

- Two bytes of error code.

[3]For example, CAN delivers 13 bits of the MTI via each frame (the special bit is known to be zero).

- Two bytes of MTI. If the frame transport only delivered part of the MTI[4], that content is returned with the rest of the MTI bits set to zero.
- Any extra bytes that the node wishes to include. There can be zero or more of these. These must be described in the node documentation.

Nodes must process this message even if not all of the contents are provided.

Error codes: *(combine the errors flags into a single section)*

0x1000 bit: if set, this is known to be a temporary error, and the interaction can be retried.

0x2000 bit: if set, this is known to be a permanent error, and the interaction must not be retried.

2.5.6 Protocol Support Inquiry

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Protocol Support Inquiry	Y	N	N	0x32E0	(none)

2.5.7 Protocol Support Reply

Name	Dest ID	Event ID	Simple Node	Common MTI	Data Content
Protocol Support Reply	Y	N	N	0x32F0	One or more bytes identifying the supported OpenLCB protocols; see Table immediately below for coding.

A 1 bit in a position indicates that the corresponding protocol is supported by the sending node. A 0 bit in a position indicates that the corresponding protocol is not supported by the sending node.

[4]For example, CAN delivers 14 bits of the MTI via each frame (the special bit is known to be zero and the stream/datagram bit can be inferred).

vv vv vv vv vv vv	Protocol
0x80 00 00 00 00 00	Simple Protocol subset
0x40 00 00 00 00 00	Datagram Protocol
0x20 00 00 00 00 00	Stream Protocol
0x10 00 00 00 00 00	Memory Configuration Protocol
0x08 00 00 00 00 00	Reservation Protocol
0x04 00 00 00 00 00	Event Exchange (Producer/Consumer) Protocol
0x02 00 00 00 00 00	Identification Protocol
0x01 00 00 00 00 00	Teaching/Learning Configuration Protocol
0x00 80 00 00 00 00	Remote Button Protocol
0x00 40 00 00 00 00	Abbreviated Default CDI Protocol
0x00 20 00 00 00 00	Display Protocol
0x00 10 00 00 00 00	Simple Node Information Protocol
0x00 08 00 00 00 00	Configuration Description Information (CDI)
0x00 04 00 00 00 00	Traction Control Protocol (Train Protocol)
0x00 02 00 00 00 00	Function Description Information (FDI)
0x00 01 00 00 00 00	~~DCC Command Station~~Traction Proxy Protocol
0x00 00 80 00 00 00	Simple Train Node Information Protocol
All others	Reserved for future protocol bits. Shall be sent as 0 and ignored upon receipt.

2.6 Interactions

All nodes must be able to take part in all standard interactions.

2.6.1 Node Initialization

Newly functional nodes, once their start-up is complete and they are fully operational, shall send an "Initialization Complete" message and enter Initialized state.

- There is no guarantee that any other node is listening for this. No reply is possible.
- Nodes must not emit any other OpenLCB message before the “Initialization Complete” message.

Sending the IC message is required to insure that higher-level tools are notified that they may start to work with the node.

2.6.2 Node ID Detection

Upon receipt of a Verify Node ID Number message addressed to it, or an unaddressed Verify Node ID Number message, a node will reply with an unaddressed Verified Node ID Number.

If a node receives multiple Verify Node ID Number messages before being able to reply, it may combine multiple unaddressed Verified Node ID number responses into one.

This can be used as check that a specific node is still reachable. When wire protocols compress the originating and/or destination NID, this can be used to obtain the full NID.

The standard Verify Node ID Number interaction can be used to get the full 48-bit NID from a node for translation. At power up each node must obtain a alias that is locally unique. Gateways will also have to obtain unique aliases for remote nodes they are proxying on to the segment.

(With the addition of an addressed form of the Verified Node ID message, is the above complete?)

2.6.2.1 Example: Node obtaining local alias from full node ID

For wire protocols like CAN, which require short node aliases to send messages, Verify Node ID can be used to get the CAN alias from a known node ID by sending the global form with the full node ID in the data. Only the desired node will reply, that reply can be identified by the full node ID in the data part of the message, and the desired node's alias will be in the source ID part of the message.

2.6.2.2 Example: Finding a full node ID from a local alias

For wire protocols like CAN, which require short node aliases to send messages, Verify Node ID can be used to get the full node ID from a local alias by sending the addressed form to the alias. Only the desired node will reply, that reply can be identified by the known alias in the source ID part of the message, and the full node ID will be in the data part of the message.

2.6.2.3 Example: Finding all nodes

Send the global form with no node ID.

2.6.2.4 Example: Confirming that a specific node can still be reached

If you have the necessary information (e.g. node alias) use the addressed form, as it's less load on the entire system. Otherwise use the global form with the Node ID in the data.

2.6.3 Error Handling

There are multiple mandatory error-handling scenarios defined.

(Need to explain “optional” here)

2.6.3.1 Reject Addressed Optional Interaction

- Node A receives an addressed message from Node B that carries Node A's NID.
- The MTI indicates the start of an optional interaction.
- If Node A does not want to take part in the optional interaction, it may send an Optional Interaction Rejected message addressed to Node B with the original MTI in the message content. There is no requirement that OIR be sent; the node may silently ignore the incoming message.

(The message content also contains an optional reason code and an optional data value. (Define use))

(This is written that sending the OIR in return is optional. This greatly increases the complexity of error handling on the originating node, though, as it can't assume that lost messages are a transient error. It would be better to have this be a mandatory response to simplify that. Does it add much complexity cost to the nodes?)

(The phrasing is also in terms of an "optional interaction", which doesn't cover the case of "undefined interaction", e.g. an MTI that's not allocated. Since any given node doesn't actually know whether an MTI has been allocated or not (it might have been built a while ago, with new optional protocols added since then), this should be rephrased. I'll add a to-do item for that too.)

Example: A CAN node receives an addressed message with an unrecognized MTI:

19EDCAAA 0123

It replies with

19068123 0A AA 20 00 0E DC

The 0C indicates Reject Optional Interaction. The 0E DC is as much as can be reconstructed of the original MTI. Note that it includes an addressed bit set, 10 00. The 20 00 is the error code, which in this case indicates this is a permanent error.

2.6.3.2 Reject Unaddressed Optional Interaction

- Node A receives an unaddressed message from Node B.
- The MTI indicates the start of an optional interaction.

If Node A does not want to take part in the optional interaction, it silently drops the message without reply.

2.6.3.3 Reject Addressed Standard Interaction Due to Error

- Node A is taking part in an addressed interaction with Node B. Either node may be able to send the next message.
- Some error condition prevents Node A from continuing the interaction.
- To terminate the interaction, Node A sends a Terminate Due to Error message to Node B. It then resets it's state so as to no longer be taking part in the addressed interaction.

The message content contains the most recent MTI received in this interaction, a mandatory reason code and an optional data value. The use of these fields is to be defined, but reserved space is to be transmitted, so we specify the bytes.

Note that the specification doesn't say whether Node A or Node B started the interaction. It could have been either. Node A is just the name for the node that can't continue and wants to stop the interaction.

This is a very coarse mechanism that is meant to handle rare events that should not routinely occur. The “most recent MTI received” values is not always sufficient to determine which interaction is being referred to. Higher level protocols should define more focused and reliable mechanisms if they are likely to encounter errors.

Nodes must handle messages of this type that arrive without MTIs and error code information.

Needs definition of a permanent vs temporary bit in the error code information (consider choosing bits in a similar way to the datagram definitions)

(add an example e.g. two PIP requests which the node can't handle, please retry) (Make it clear when this option is not available in the protocol docs; assume it is if not otherwise indicated; simplifies little nodes)

2.6.4 Duplicate Node ID Discovery

OpenLCB nodes must have unique node IDs. The Frame Transfer protocol will detect duplicate node IDs on a single CAN OpenLCB segment, e.g. a single CAN bus, but is not intended to detect duplicate node IDs across multiple segments. To detect duplicates across the entire connected OpenLCB, all OpenLCB nodes must indicate an error if they detect an incoming message with a Source Node ID equal to their own. If possible, they should indicate it at the board itself using a light or similar. If possible, they should emit a PCER message with the “Duplicate Source ID detected” global event, which will carry the duplicate node ID in the Source Node ID field. (But how can they “must” if those are “if possible”?)

After sending the “Duplicate Source ID detected” global event, the node should not transmit any further “Duplicate Source ID detected” messages until reset because this message will be received at the other duplicate-ID node(s), resulting in additional “Duplicate Source ID detected” global events and causing a possible message loop. (Optionally, could allow to send again after e.g. 5 seconds)

Yes, this is level jumping, but it's the best way to do it within the existing structure. *(reference Event documents for more detail)*

To further improve the reliability of this detection, OpenLCB nodes should, but need not, emit a Verified Node ID message every 30 to 90 seconds. As an implementation detail, it's recommended that CAN-attached nodes use their NIDa to pick that interval so that messages don't bunch up.

2.6.5 Supported Protocol Inquiry and Response

OpenLCB defines various optional protocols. If another node attempts to use a protocol that the target node doesn't implement, there are well-defined rules for how the target node will either signal an error or ignore the request.

For some uses, it's convenient to be able to tell whether a node implements a protocol before attempting to use it. The Protocol Support Inquiry and Protocol Support Reply messages define a method for doing that.

To determine which protocols a node implements, a Protocol Support Inquiry message is sent to the specific node. It will reply with a Protocol Support Reply message that contains six bytes of data. A specific bit position has been reserved for each defined protocol. If the bit is zero or not present, the protocol is not supported and requests to use it will result in a error. If present and 1, the protocol is supported.

It is not necessary to use these messages to check whether an addressed protocol is supported by a node before attempting to use the protocol. If it's not, the standard error handling mechanism will indicate that.

These messages provides a way to check, without errors, whether the protocol is supported. Avoiding errors provides a cleaner system. Further, this method can check support for protocols that use global (non-addressed messages); nodes are not permitted to return errors for global messages.

We don't explicitly reference all the protocol definitions that are associated with specific bits. Implementors can find them from the protocol names, and we don't want to clog up this section of the Standard with a set of references that we'll have to update continually.

The CAN messages were defined to be part of the “simple” subset. Low-end nodes may want to implement this protocol so that higher-function nodes can easily learn their limitations.

Generally, the node designer will just provide a simple fixed value for the reply. On CAN, the entire frame is fixed except for the source and destination aliases. The destination alias can be taken directly from the request packet.

The requirement for messages of “one or more bytes” is to allow for future expansion. Messages of more than 6 bytes on CAN will be sent using the start/end message framing bits. It's not necessary to send extra zero bytes past any bits that need to be asserted.

Other nodes can snoop on this interaction to learn the protocols supported by a node, without having to send their own inquiry. It's not expected that the protocols supported by a specific node will change with time, although the Standard does not require that they be immutable. There's no mechanism for a node to request an update if another node's list of supported protocols changes, nor is there a mechanism for a node to inform others that its list has changed.

A node must promptly reply to the Inquiry message with the Reply message. On the other hand, the general message standard requires that a node not supporting this protocol reply to the Inquiry message

with a Optional Interaction Rejected message. Either way, a node sending a Protocol Id Inquiry message can count on getting a rapid reply.

OpenLCB is big-endian, so the protocol bits have been assigned from the MSB of the 1st byte.

In general, nodes using this interaction don't need to know the meaning of bits defined after the node was created. A node is looking to see whether a particular protocol is present or not so that it can select what actions to take. A newly-defined protocol isn't among the things that the node will try to reason about. This allows us to eventually extend the length of the reply without causing trouble for existing nodes.

We define the bits here, rather than in the individual protocol definitions, to reduce the risk of duplicate assignments. Duplicate assignments would be obvious here, but not so much when spread across separate documents.

Before the CAN message start/end bits were defined, the 0x00 00 00 00 00 0F bits in the PIP reply message were defined for expansion. Nodes were to ignore any frames with one or more of those bits set. That mechanism has been deprecated and all known implementations updated to remove it. It's possible that nodes implementing that early version will misinterpret PIP replies once those bits represent specific protocols, but bits in the first CAN frame are valuable, so rather than reserve bits for that unlikely case, we've just required that those (few) nodes be fixed.

This interaction is optimal for requesting information from a single node, but not for requesting which nodes on the entire network can provide a service. Event-based methods, whether a capability is announced via an event or the Identify Producer and Identify Consumer mechanism is used to find which nodes can source the event that identifies the capability, may be more effective for that use case.

The information returned is intended to be considered static: A node may request it and never have to request it again, because it won't change. (For development purposes, a node might be reprogrammed, so it might be useful if configuration tools have a way to force rescan of this data)

2.7 Routing

All messages can be, but don't have to be, presented to every node for processing.

All components that forward messages may, but are not required to, forward every message. They are required to forward addressed messages in such way that the message arrives at the addressed node, and that messages to that node preserve order within a given priority, and that a lower priority message never passes a higher priority message.

Unless specific arrangements are made within a specific protocol, all components that forward messages are required to forward unaddressed (global) messages in such a way that the messages arrive at every node. (The simple node protocol provides an example of a specific arrangement for not forwarding some messages)

Global messages must be presented to and acted upon by the node sending them. For example, a message that requires global replies from all notes with a certain condition must result in a reply message from the original sending node if that condition is true for that node. (This enables passive condition monitoring by other nodes on the network)

(Need to talk about buffering here. Switching between transports is a routing issue, and has to be done right, although much of it will be specified in data-link level standards)

CAN implementations must send the frames of a message together to reduce buffering. Although higher-priority messages can be sent in the middle of a lower-priority message, there are only N priority levels, so receives only need to provide N buffers per node to reassemble CAN messages.

2.8 Delays and Timeouts

Nodes shall send messages required by OpenLCB protocols within 750 msec unless otherwise indicated in the documentation for the specific protocol interaction.

Nodes may, but are not required to, use a timeout mechanism to protect against messages lost due to malfunctions. Such a timeout shall not be shorter than 3 seconds.

Needs a discussion of handling failed communications. The protocols are all designed to have error-responses (error reply to datagram; Option Interaction Rejected) instead of silently failing for directed messages. Timeout logic is only necessary to handle transport failures and failures of nodes to execute the protocols, including e.g. when they power off in the middle of something. The Message Network Standard should say that "Nodes shall allow at least nnn msec for a reply to be received to their communications" and "Nodes shall reply to messages within mmm msec" (See above). Then ~~this~~the Message Network TN can talk about timeouts, how you have to assume that you'll have to wait for a reply when arranging state machines and buffering, and the generic issues around recovering from an expected reply not being received. Any specific issues with timeout recovery can be discussed in the message-specific parts of the TN.

3 Simple Node Protocol Subset

OpenLCB uses the "Simple Node Protocol" concept to distinguish a subset of global message types that are never needed by certain "simple" nodes. They can then be rapidly ignored by those nodes, gateways can filter them out, etc. This note describes the Simple Node Protocol. It is not normative on node implementors.

Simple nodes are defined as the leaf nodes that do basic layout control (input/output) operations. These nodes need to be able to[5]

- indicate their presence (by sending Initialization Complete messages and replying to Verify Node messages)
- send and receive event reports (through PCER messages and associated reports)
- be configured (through related messages, datagrams and in some cases streams)

and perhaps other things in the future. Those messages form the Simple Node Protocol subset of the full OpenLCB protocols. Note that this is an asymmetric subset: Simple nodes can send some types of messages and receive others.

[5]Simple nodes also have to do whatever is needed to function with their specific wire protocol, e.g. send CID/RID frames on CAN.

On the other hand, gateways, configuration tools and other network-aware nodes are not simple nodes. These nodes need access to the full OpenLCB protocol so they can

- learn about the appearance of other nodes (by receiving Initialization Complete messages)
- learn about other producers and consumers (by receiving status messages)

and similar. To do this, they must be able to send and receive every message type.

For ease of filtering, a specific bit in the MTI identifies the global messages needed by simple nodes. This bit allows OpenLCB to define new MTIs in the future and still include them in the "simple node protocol" subset or outside it without having to modify existing nodes. See the MTI allocation TN for more information on this.

Gateways that are serving network segments (e.g. single CAN segments) that contain only simple nodes may suppress unaddressed (global) messages that do not contain simple-node MTIs.

3.1 Protocol Description

Operationally, the simple node protocol is defined by the MTIs that carry a set Simple Node bit, plus all addressed messages. This section summarize received transmitted messages, and describes the reasoning behind those choices in the current MTI definitions.

3.1.1 Messages Transmitted

Simple nodes may transmit any message, which must propagate correctly.

3.1.2 Messages Received

Simple nodes must receive any message specifically addressed to them, plus the following unaddressed global messages:

- Verify Node ID – They need to receive this so that they can reply to it.
- Verified Node ID – They need to receive this because it's the reply to their own request, which might be used to e.g. locate a node for delayed sending of status
- Protocol Support Inquiry – They need to receive this so that they can reply to it.
- Identify Consumers, Identify Producers, Identify Events – because others will ask this of them
- Learn Event – so they can be programmed
- P/C Event Report – what they do for a living

In the future, additional MTIs will be defined. If simple nodes need to received them, the MTI will indicate that via the Simple bit; see previous section.

3.1.3 Messages Not Received

Messages not listed in the section above do not need to be received by simple nodes.

A brief description of why the following message types are not necessary for simple nodes:

- Initialization Complete: Used to indicate that a node is newly available to the network. Simple nodes only care about their specific tasks, and by definition are not interested in the overall structure and availability of the network.
- Consumer Identified, Consumer Identify Range: These are of interest to gateways and configuration tools, but an individual producer does not need to know which (if any) nodes are consuming its produced events.
- Producer Identified, Producer Identify Range: These are of interest to gateways and configuration tools, but an individual consumer does not need to know which (if any) nodes are producing its consumed events.

4 Gateway Processing

Idea is to gather all the info on gateway processing here; TN-level discussion of distributed data-link level state machines (CAN alias handling at the remote side of the gateway); routing of global and addressed traffic; desire to have addressed traffic optionally throughout the network for snooper support; distributed routing based on InitializationComplete messages identifying location; handling of nodes that move from segment to segment e.g. throttles.

5 Expansion

5.1 Longer MTIs

On the expansion past 16 bit MTIs both for global and addressed messages. Easy to take up another byte, harder to get optimal use out of it. To preserve the CAN priority structure, an MTI with each of the priority levels has been reserved. Messages with one of these MTIs will treat the start of the data second of the message (after the destination address et al, if present) as another byte of MTI. Nodes created before the expanded MTIs need to be defined can just ignore them.

5.2 Longer Messages

Longer than 8-byte global, 7-byte addressed messages don't fit in a single frame. Discuss mechanism. Includes issues of guaranteed place for global to land in a node, e.g. memory management. (Spare frame type; request resend w/o buffering, but how long to hang on to original?)

PIP reply uses “break into N frames with MTI at start”: One message becomes N smaller ones. But it's a directed reply, so the requestor can ensure that the response will be processed on receipt. Label end with a non-full CAN frame (though Io code sends those in the middle), if needed a zero-length frame.

How do multi-frame addressed messages work if the receive buffer is full and the receiver has to discard the frames? There are several possible approaches. The datagram protocol and stream protocol demonstrate two approaches: reject & retransmit, and negotiate a buffer in advance. There are certainly other ways to handle it too.

For example, the SNII protocol sends a short message and gets back a lot of data. The SNII reply uses the start-end bits so that the individual CAN frames become one long message. The node that initiates the SNII exchange _knows_ that a reply is coming back, _knows_ the max size of that, so should make

sure that it has the buffer available (or can handle it some other way) when the large message comes back.

A CAN node may only send higher priority frames while in the middle of sending the frames that make up a long message. It may not send messages with the same or lower priority.

Need a few words about gateways: Buffer management is an issue (though not that big an issue, since a CAN link can only provide 8kB/s of data in any case) Nodes should send only one fragmented message at a time to reduce buffer use (though separate nodes can clearly interleave). This situation is greatly simplified because the gateway is assembling onto single messages on the non-CAN side, and only receiving single messages (no need to assemble them from pieces), so the buffer management can be handled separately on the two sides.

Expansion: PIP example. CAN-based code looks at 1st frame. To allow expansion, has to already look at first frame, reject/ignore if later frame.

The idea of idempotent messages and replies, which simplifies protocols that can reject a message due to buffer full. (Just resend it, no book-keeping needed)

Nodes shall not send short (less than 8 byte content) frames without the “last” bit active.

Because most frames are “only”, to simplify reading the MTI bytes, we've defined the first and last bits as active-0.

How to handle expansion? E.g. what must nodes check? Reject if final not active, but that isn't expandable. Consider:

- first, last, expected content – handle as expected

Future expansion within the 1st frame, something already envisioned:

- first, last, expected content + more in frame – handle original content as expected

Future expansion to larger message via additional frames:

- first, not last, expected content + more in frame – handle content
- not first, not last – more content, you've already replied, ignore
- not first, last – end of more content, you've already replied, ignore

This works, but the reply or response (if any) goes back before the full set of frames has come in on the wire. Nodes must be able to handle that.

(What we're trying to avoid here is nodes being required to have a buffer large enough to hold any possible future expansion of the message size, because that's a burden of the future on nodes now. Fixing a max size is only a partial answer to this)

An alternate expansion is to just save the original info in a buffer that only holds that original number of bytes, and then execute it when the final bit is received. To enable this, we require that any node send the frames of a single message consecutively, to prevent needing buffers for multiple simultaneous messages. That can still happen with multiple nodes sending. (In theory, CAN arbitration & the message structure would make it more likely that only one node would send the consecutive frames, as

they all have the same header; but a higher priority message from another node can start in the middle of transmission, and we can't require than node implementations be fast enough to send bit-sequential messages)

Gateways need to accumulate frames into single messages. They might need a large amount of buffering for this, in the limit one message buffer per CAN node being served (e.g. up to 50 or N*50 with electrical repeaters).

Does there need to be a permanent restriction on the max size of a message to reduce the size of needed buffers and to prevent stalling of e.g. CAN segments? Same size as datagram, which already has the issue? Note that SNII is now described as up to a total of 253 data bytes. (SNII could also send its reply as two smaller messages) Fifty buffers of 256 bytes is only 13kB, not so much. We've imagined that gateways may be sold as “serving up to N nodes”, in case people want to do advanced things like mixing repeaters, bridges and gateways. Somebody just needs to make a gateway that goes to 20*50...

PIP request example:

The combination of a Start bit and Stop bit is the entire message. If those are in two separate frames, you should still only receive one reply. But that reply can be to the frame with the start bit. There's no requirement that a node receive the entire message before replying.

PIP request, where there's no real data, is a good example of this. PIP may be extended to have (optional) data later on, so nodes should be able to receive-and-ignore extra data. But there's no reason for a current node to allocate a (big?) buffer to handle that assembly of PIP frames for future expansion. Instead, they can just process what they receive when they've received enough of it (e.g. the 1st frame in this case), and just know to skip the rest. So the Olcb* implementation does just that: Reply to frame with start bit set, whether or not stop bit set; ignore PIP frames without start bit set.

5.2.1 Long global messages

OpenLCB-CAN does not support global messages with data content longer than eight bytes. This section is a discussion of the reasoning behind that, and some of the ways that the restriction could be relaxed.

There are two problems: Bits for coding start/end, and buffer management. The bits are just a matter of deciding to use MTI space. Buffer management is harder.

In general, use of long global messages would require unbounded buffer space in the receiving nodes. OpenLCB is a distributed, peer-to-peer system where an arbitrary number of nodes may chose to invoke a specific function at any given time. The huge layout case may involve hundreds of nodes doing the same thing on an sensible time scale. If that involves long-global messages, their sequences of CAN frames may be intermixed on the CAN bus.

- Gateways that move the long-global message onto the CAN bus can be required to send one message entirely before starting another, but that doesn't help with priority-driven mixing of messages from nodes directly on the CAN bus segment.
- One could restrict the design so that only one specific node on a layout can use a long-global message, but that raises a number of complexities that don't currently seem worth it just to get long-global messages.

- Specifying the maximum length of the long-global message helps reduce the size of the needed buffers, though not their number. This could be specified either absolutely (for all messages, useful for gateways that might not understand the content of the message) or for a specific long-global MTI (so that implementing end nodes can reserve the minimum size necessary).
- Rejecting messages that arrive without local buffer space makes the original global operation complicated because either an addressed or global retry of the operation is likely to be complex and error prone.

There are also good alternatives:

- Manually send the large message as smaller ones, defined in the specific protocol being implemented. This is ugly, because it moves a CAN limitation out into the larger OpenLCB, but it does the job for now.
- Use a short global to get a reasonable-size list of relevant nodes, which are then send addressed messages. This generates more on-segment traffic because it doesn't use the multi-point nature of global messages, but makes buffer management (including retries) straight-forward.

Still, if the decision was made to provide long globals, it could be done. For buffer management, you'd allow a node to reject the global message with a temporary error code. That carries the Node ID of the rejecting node (as an alias on CAN), so that originator of the request can resend the same request as an addressed message to the rejecting node(s). Retries by resending the global request should be forbidden, because they are likely to not converge as some other node runs short of buffers. The higher-level protocol that uses this should, if possible, cue the node to expect the global so that it can reserve buffer space. Etc.

To find the space for the start/end flags, there are several options. Global messages (destination present bit reset) with the Event ID present bit reset have the two modifier bits available as start-end flags. (The Simple bit might also be involved in this)

- Excluding the MTIs with the 'Event ID present' bit set is a tradeoff. By doing this, the high-volume messages used to identify producers and consumers of events can fit in a single frame. We could fix this by (for the 2nd time) redefining the MTIs for those messages to use higher bits in the MTI. Or leave it like it is.
- Dedicating those two bits (for global messages types without event IDs) takes away some MTI space for those specific types. That's a tradeoff between future expansion (gateways knowing how to handle MTIs they haven't seen yet) and number of available MTIs. For example, the Identify Events request is unlikely to ever grow to more than one frame. Having those two bits in the MTI be occupied anyway reduces the total number of possible MTIs. But it does allow the addition of longer-than 8 bytes global messages that can pass through current-production gateways.

Another approach would be to define specific MTIs for start/middle/end of long messages. But this is problematic for those messages once off the CAN bus, and if the bits for this are not specifically allocated (e.g. like this proposal), it makes automatic disassembly and assembly of messages to frames in gateways much harder. And it's a complete mess if e.g. another transport comes along that has a 14 byte limit.

6 CAN Adaptations

6.1.1 Initialization Complete

Indicates that the sending node initialization is complete, and once the message is delivered, reachable on the network.

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Initialization Complete	N	N	N	0x0100	0x1910,0sss	Sending Source ID

6.1.2 Verify Node ID

Issued to determine which node(s) are present and can be reached.

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Verify Node ID	N	N	Y	0x28A7	0x1949,0sss	(optional) Node ID
	Y	N		0x30A0	0x1948,8sss fddd	(optional) Node ID

There are multiple forms of the Verify Node ID message.

The global (unaddressed) format may include an optional NodeID. If present, only nodes with a matching Node ID should reply. If absent, all nodes should reply.

The addressed form may include an optional full NodeID in the data section. The addressed node must always reply, whether or not a Node ID is carried in the data, and whether there's a match when the optional Node ID is present. The optional NodeID idea just serves as documentation of the intent of the request. (Makes sense on CAN, not so much on other transports that don't use aliases).

(Should the discussion of replying be here, or under interactions? Might be better to have this section talk about the meaning of the Node ID, rather that about replies.)

(In the TN, add an example of the full format messages, which can include two copies of the Node ID due to the way destination alias mapping works)

6.1.3 Verified Node ID

Reply to the Verify Node ID message.

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Verified Node ID	N	N	Y	0x28B7	0x1917,0sss	Full Node ID of sending node

The node ID in the data is redundant on wire protocols that carry the full source ID, but can be very valuable for wire protocols that abbreviate ("alias") the source ID within the messages, e.g. CAN.

6.1.4 Optional Interaction Rejected

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Optional Interaction Rejected	Y	N		0x30C0	0x1906,8sss fddd	Error codes, MTI, optional information

The contents are, in order:

- Two bytes of error code.
- Two bytes of MTI. If the frame transport only delivered part of the MTI[6], that content is returned with the rest of the MTI bits set to zero.
- Any extra bytes that the node wishes to include. There can be zero or more of these. These must be described in the node documentation.

Nodes must process this message even if not all of the contents are provided.

Error codes:

0x1000 bit: if set, this is known to be a temporary error, and the interaction can be retried.

0x2000 bit: if set, this is known to be a permanent error, and the interaction must not be retried.

6.1.5 Terminate Due to Error

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content

[6]For example, CAN delivers 13 bits of the MTI via each frame (the special bit is known to be zero).

Terminate Due to Error	Y	N		0x30D0	0x190A,8sss fddd	Error code, MTI, optional information

The contents are, in order:

- Two bytes of error code.
- Two bytes of MTI. If the frame transport only delivered part of the MTI[7], that content is returned with the rest of the MTI bits set to zero.
- Any extra bytes that the node wishes to include. There can be zero or more of these. These must be described in the node documentation.

Nodes must process this message even if not all of the contents are provided.

Error codes: *(combine the errors flags into a single section)*

0x1000 bit: if set, this is known to be a temporary error, and the interaction can be retried.

0x2000 bit: if set, this is known to be a permanent error, and the interaction must not be retried.

6.1.6 Protocol Support Inquiry

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Protocol Support Inquiry	Y	N	N	0x32E0	0x1Edd,dsss 2E	(none)

6.1.7 Protocol Support Reply

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Protocol Support Reply	Y	N	N	0x32F0	0x1Edd,dsss 2F	One or more bytes identifying the supported OpenLCB protocols; see Table in general description.

[7]For example, CAN delivers 14 bits of the MTI via each frame (the special bit is known to be zero and the stream/datagram bit can be inferred).

Table of Contents

755