



## OpenLCB Working Note

### OpenLCB Message Network

Sep 24, 2012

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

- 5 This explanatory (future Technical Note) contains informative discussion and background for the (eventual) corresponding “OpenLCB CAN Message Network Specification”.

10 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.

## 15 2 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.

### 2.1 Introduction

20 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.

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

### 2.2 Intended Use

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

30 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.

## 2.3 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:

- 35
- 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:

- 40
- 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.

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

## 2.4 Message Format

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

50 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.

### 2.4.1 Message Type Indicators

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

55 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.

60 This section describes how the numeric values for those MTIs are allocated. The current allocations are documented in a separate spreadsheet<sup>1</sup>. 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

<sup>1</sup>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.

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.

65 *(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.

70 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.
- 75 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”.

80 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.
Stream or Datagram	1 bit 0x1000	1 means stream or datagram, 0 means regular message.
Static Priority	2 bits 0x0C00	0 to 3 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.

85 The next bit, the “Dest ID flag” in the least significant bit of the top nibble, 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.<sup>2</sup>

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

90 The contents of the top nibble is still under development. Note that there are only four values currently defined as useful: 2 and 3 for unaddressed and addressed general purpose messages that will route to any CAN network; and 0 and 1 for unaddressed and addressed 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 (and very likely the Dest ID flag bit) will be preserved. Values 4-15 or the top 2  
95 bits are not currently used and available for the future.

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

- The top 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  
100 delivered to a node should be processed.

<sup>2</sup>The “simple node protocol” is an exception to this, which needs to be worked into this Standard.

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

- The first bit (MSB) in the nibble indicates this message carries a P/C Event ID field when set to 1. Setting 0 means that the 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.

#### **2.4.1.1 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.

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.

- 140 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**

- 145 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)

#### **2.4.1.1.1 Addressed and Global CAN MTIs**

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

- 150 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 1<sup>st</sup> two bytes of the data part of the CAN frame. The top nibble of the 1<sup>st</sup> byte contains flags (see below); the lower nibble of the 1<sup>st</sup> byte, and the entire 2<sup>nd</sup> byte contain the 12-bit destination alias from the CAN frame level protocols.

- 155 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

- 160
- 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.

165

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

**Table 3: Unaddressed MTI CAN Header Format**

#### 2.4.1.1.2 Datagram and Stream frame format

170 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.

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

Table 4: Addressed MTI CAN Header Format

175

#### 2.4.1.1.3 CAN Notes

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

180 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)

#### 185 2.4.2 Message Content

The message content consists of:

- The source Node ID
- The MTI
- If flagged as present, the destination Node ID
- 190 • 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.

## 2.5 States

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

- Uninitialized
- Initialized

Nodes shall start in the Uninitialized state.



200 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.

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

## 2.6 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.

210 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.

### 2.6.1 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

215

### 2.6.2 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.

220 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.

225 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 than 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)

### 230 2.6.3 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.

235 (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.

### 240 2.6.4 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:

- 245
- Two bytes of error code.

- Two bytes of MTI. If the frame transport only delivered part of the MTI<sup>3</sup>, 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.

250 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.

### 2.6.5 Terminate Due to Error

Name	Dest ID	Event ID	Simple Node	Common MTI	CAN format	Data Content
Terminate Due to Error	Y	N		0x30D0	0x19A0,8sss fddd	Error code, MTI, optional information

255

The contents are, in order:

- Two bytes of error code.
- Two bytes of MTI. If the frame transport only delivered part of the MTI<sup>4</sup>, 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.

260

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.

265

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

### 2.6.6 Protocol Support Inquiry

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

15 <sup>3</sup>For example, CAN delivers 13 bits of the MTI via each frame (the special bit is known to be zero).

<sup>4</sup>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).

Protocol Support Inquiry	Y	N	N	0x32E0	0x1Edd,dsss 2E	(none)
--------------------------	---	---	---	--------	----------------	--------

## 270 2.6.7 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	Six 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.

275

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

vv vv vv vv vv vv	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 00 00 00 00 0F	Reserved. Shall be sent as 0 and ignored upon receipt.
All others	Reserved for future protocol bits. Shall be sent as 0 and ignored upon receipt.

## 2.7 Interactions

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

### 2.7.1 Node Initialization

280 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.

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

### 2.7.2 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.

290 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.

295 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?)

### 2.7.2.1 Example: Node obtaining local alias from full node ID

300 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.

### 2.7.2.2 Example: Finding a full node ID from a local alias

305 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.

### 2.7.2.3 Example: Finding all nodes

Send the global form with no node ID.

### 310 2.7.2.4 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.

## 2.7.3 Error Handling

There are multiple mandatory error-handling scenarios defined.

315 (Need to explain "optional" here)

### 2.7.3.1 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.

320

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

325 (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?)

330 (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:

19EDCAA 0123

It replies with

335 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.

### 2.7.3.2 Reject Unaddressed Optional Interaction

- 340
- 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.

### 2.7.3.3 Reject Addressed Standard Interaction Due to Error

- 345
- 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.

- 350
- 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.

- 355
- 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.

- 360
- 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)



#### 365 **2.7.4 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”?)

375 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*)

380 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.

#### **2.7.5 Supported Protocol Inquiry and Response**

385 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.

395 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 an 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.

400 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.



- 405 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.

- 410 The requirement for messages of “six 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.

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 the list of supported protocols changes.

415

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.

- 420 OpenLCB is big-endian, so these have been assigned from the MSB of the 1<sup>st</sup> 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 in any one of several ways.

- 425 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 where to ignore any frames with one or more of those bits set. That mechanism is no longer present, but some of those nodes exist, so for backwards compatibility those bits are permanently reserved.

430

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.

435

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)

## **2.8 Delays and Timeouts**

- 440 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 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”. Then this 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<sup>5</sup>

- 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.

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.

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

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

480 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

485 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
- 490 • 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

495 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:

- 500 • 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.
- 505 • 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 Expansion

### 4.1 Longer MTIs

- 510 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. Expansion needs to be indicated by something that's in the CAN section, so use all zero.

### 4.2 Longer Messages

- Longer than 8-byte global, 7-byte addressed messages don't fit in a single frame. Discuss mechanism.
- 515 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.

520

How do multi-frame addressed messages work if the receive buffer is full and the receiver has to discard the frames?

The datagram protocol and stream protocol show how it can work. There are certainly other ways to handle it too.

- 525 For example, the SNII protocol sends a short message and gets back a lot of data. Previously, those were just a series of independent messages. SNII can still be like that, but it'll be better to add the start-end bits to those messages 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
- 530 message comes back.

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).

- 535 Expansion: PIP example. CAN-based code looks at 1<sup>st</sup> 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)

540

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.

545 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 1<sup>st</sup> frame, something already envisioned:

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

Future expansion to larger message via additional frames:

- 550
- 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.

555 (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)

(PIP or SNII could be an example here)

560 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)

565

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  $N \times 50$  with electrical repeaters). Does there need to be a permanent restriction on the max size of that buffer?

570 Same size as datagram, which already has the issue? Note that SNII is now described as up to a total of 253 data bytes. 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  $20 \times 50$ ...

575 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.

580 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.

585

DRAFT

## Table of Contents

1 Introduction.....	1
2 Annotations to the Specification.....	1
2.1 Introduction.....	1
2.2 Intended Use.....	1
2.3 References and Context.....	2
2.4 Message Format.....	2
2.4.1 Message Type Indicators.....	2
2.4.1.1 CAN MTI Considerations.....	5
2.4.1.1.1 Addressed and Global CAN MTIs.....	6
2.4.1.1.2 Datagram and Stream frame format.....	7
2.4.1.1.3 CAN Notes.....	8
2.4.2 Message Content.....	8
2.5 States.....	8
2.6 Definition of Specific Messages.....	9
2.6.1 Initialization Complete.....	9
2.6.2 Verify Node ID.....	9
2.6.3 Verified Node ID.....	10
2.6.4 Optional Interaction Rejected.....	10
2.6.5 Terminate Due to Error.....	11
2.7 Interactions.....	11
2.7.1 Node Initialization.....	11
2.7.2 Node ID Detection.....	12
2.7.2.1 Example: Node obtaining local alias from full node ID.....	12
2.7.2.2 Example: Finding all nodes.....	12
2.7.2.3 Example: Confirming that a specific node can still be reached.....	12
2.7.3 Error Handling.....	12
2.7.3.1 Reject Addressed Optional Interaction.....	12
2.7.3.2 Reject Unaddressed Optional Interaction.....	13
2.7.3.3 Reject Addressed Standard Interaction Due to Error.....	13
2.7.4 Duplicate Node ID Discovery.....	14
2.8 Delays and Timeouts.....	14
3 Simple Node Protocol Subset.....	15
3.1 Protocol Description.....	16
3.1.1 Messages Transmitted.....	16
3.1.2 Messages Received.....	16
3.1.3 Messages Not Received.....	16
4 Expansion.....	17
4.1 Longer MTIs.....	17
4.2 Longer Messages.....	17