	OpenLCB Technical Note
	OpenLCB-CAN Datagram Transport
Jan 24, 2013	**Preliminary**

1 Introduction

This Technical Note contains informative discussion and background for the corresponding "OpenLCB Datagram Transport Specification". This explanation is not normative in any way.

Some wire protocols can support only very short packets/frames, e.g. CAN with a limited header and data payload. OpenLCB needs to be able to move larger chunks of data than that. OpenLCB streaming provides a very large payload, at some cost in setup time and complexity. "Datagrams" move an amount of data in-between "small enough to always be atomic" and "large enough to justify streaming overhead".

The datagram protocol on CAN links uses the same structure as the common protocol, except for the need to split the message into frames.

A node may only send one datagram at a time to any given destination node. A node may interleave transmission of datagrams to separate nodes, but this is not recommended because the longer transmission window increases the probability of buffer collisions in recipient nodes.

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

2.2 References and Context

For more information on format and presentation, see:

- OpenLCB Common Information Technical Note

2.3 Message Formats

2.3.1 Datagram Content

The datagram length is coded by the transport medium: There is no separate length field in the datagram content itself.

For CAN, the general Datagram message is broken into a First frame, one or more Middle frames, and a Last frame. In the degenerate case of a very short message that can fit into one frame, there is also an Only-frame. These are described in more detail below. The CAN header values are chosen to prevent CAN priority from reordering frames in a way that would damage the transmission.

30

2.3.1.1 CAN-Datagram Content Single Frame

Note that zero-length datagrams are valid.

2.3.1.2 CAN-Datagram Content First Frame

Note that there's no requirement that this contain 8 bytes.

35
2.3.1.3 CAN-Datagram Content Middle Frame

Note that there's no requirement that this contain 8 bytes. It cannot be empty, though.

2.3.1.4 CAN Datagram Content Last Frame

Note that this frame can be zero length.

2.3.2 Datagram Received OK

40 This is an addressed message. The destination address (full Node ID or alias) can be taken directly from the received Datagram Content message.

2.3.3 Datagram Rejected

2.3.3.1 Error Codes

(These aren't the same as the Standard)

45 "Permanent error" - This node does not process datagrams, will not accept a datagram from this source, or for some other reason will not ever be able to accept this datagram. The datagram should not be retransmitted. Optionally, the node can mark the reply with one or more of several conditions:

"Invalid Datagram" - something made the datagram improper, such as longer than the max permitted length. Proper datagrams might be acceptable.

50 "Source not permitted" - datagrams from this source will never be accepted

"Datagrams not accepted" - this node will not accept datagrams under any circumstance. A node can also reject the interaction instead of sending Datagram Rejected with this code.

"Buffer shortage, resend" - The node wasn't able to receive the datagram because of a shortage of buffers. The sending node should resend at its convenience.

55 "Datagram rejected, out of order" - Should not happen, but an internal inconsistency was found in the CAN frames making up a datagram. Sender can try again if desired.

2.4 States (Normative)

2.5 Interactions (Normative)

2.5.1 Normal Transmission

60 In the base protocol a datagram is just a single message containing the Datagram MTI and the data byte(s). There is a convention for a data format indicator at the start of the data bytes, see below.

Once the datagram has been successfully received, the receiving node replies to the original source node with a "Datagram Received OK" message. Note that the node is not required to examine or process the datagram before replying; execution errors that happen later must be signaled using another
65 protocol.

2.5.1.1 CAN Protocol

The sender breaks the datagram content into one or more frames. The source NodeID alias and destination NodeID alias are contained in the header as usual. Format indicators are used to mark the segmentation into frames. The data part of the CAN frame can carry from zero to eight data bytes.

70 If there are more than eight bytes to the datagram, the rest are send as consecutive frames. Because CAN transmission retains frame order between sender and receiver, no order information is added to the frames. The first/middle/last frame is marked bin the CAN header.

There's nothing in the Standard that says that CAN frames have to be maximal size. A 9-byte datagram can be frames of 8 and 1, or 1 and 8. This is helpful for handling e.g. buffering and creation of
75 dynamic content in the source node.

The receiving node receives the frames into a buffer. The frames from more than one datagram may arrive in interleaved order, in which case the receiving node can tell them apart using their source NodeID alias and store them in separate buffers. (Any given source node can only send a single datagram at a time a single destination node) If sufficient buffers are not available, the receiving
80 node rejects the datagram and it will be resent. That reject can be sent early, so that buffer space to accumulate the datagram or even state bits to remember there is a data need not be allocated. Once the "datagram complete" indicator is received for a datagram, the node replies to the original source node with a "Datagram Received OK" or "Datagram Rejected" message.

The state machine for assembling datagram frames needs to know when e.g. the lower levels reset, so
85 that it knows to abandon a partially received datagram. It can also recover upon seeing the next start/only frame.

An interface or gateway onto a CAN link must do the datagram fragmentation and defragmentation locally. Buffer retries may be either done locally by the gateway, or by reflecting the response back to the original sender.

2.5.2 Rejected Transmission
90
Instead of "Datagram Received OK", the receiving node may return a "Datagram Rejected" message with an error code that represents the reason(s) and possible reaction(s).

2.5.2.1 CAN Protocol

There's no requirement forto a specific response to out-of-spec transmissions. If a node receives out-of-spec datagram fragments, what it does is up to it. This is because it's neither necessary nor practical to specify a unique set of logic for dealing with the semi-infinite number of possible out-of-spec transmissions. Nodes can use the various error replies to indicate whatever they know, but they don't have to.

So, a node that gets a middle-segment without a first-segment can react in various ways. It might send various replies (but it doesn't have to). It might ignore it. Etc.

There's an important thing to realize about that behaviour, though:

Getting another valid datagram sequence while receiving one (or more) that fills the local buffers is _not_ an out-of-spec condition, and the protocol defines exactly how the node is to react. There's no flexibility therein this case.

3 Background Information
3.1 Use Cases

Configuration: Simple one-value reads and writes of configuration data are one-to-one operations that need to exchange 4-8-16-32 bytes of data.

External-network control: If the OpenLCB net is attached to single external networks, e.g. a DCC or LocoNet system, one-to-one transmission can be used to exchange native commands. E.g. "Send the following LocoNet message" may need to be up to 18 bytes long with current LocoNet definitions. Communcation to LocoNet/DCC/etc can be one-to-one to insure communications; return messages can be many one-to-one to listeners, but there's also a need for one-to-many or broadcast which is not addressed here.

RFID tags carry 40 (vs 64, and need to plan for future growth) bits of payload. Adding in some status and location information, reporting a RFID tag requires a 8-16 byte payload, and that's likely to grow in the future.

3.2 Discussion

Datagrams are a one-to-one connection between nodes. They have a source address and a single destination address, and only have to be delivered to that single destination. Delivery is guaranteed if the destination node is active, initialized and reachable, but it's not possible to guarantee in advance that a buffer is available in the destination to receive and assemble the datagram.

At the high end, what's the maximum size for a datagram, beyond which it makes sense to move to a stream? The stream startup overhead is small, so that streams can be used for even small amounts of data. The maximum datagram size must be short enough that nodes can afford to have a fixed receive-and-process buffer for datagrams allocated, unlike streams that can be too large to buffer as a whole. As

a balance between these, and to optimize certain layouts on CAN links, a size of 72 bytes has been
130   selected.

At the low end, datagrams should go all the way down to zero bytes, because they are distinct from
Events.

The datagram protocol doesn't need error correction, because OpenLCB is based only on reliable links.
CAN, TCP/IP, etc, handle their own error conditions. The datagram protocol needs flow control and
135   synchronization, however, as a small node might not have resources to accumulate a large number of
datagrams that are arriving in an interleaved order. In order for the sending node to know it can free its
buffer holding the transmitted message, there needs to be an "received OK, acknowledged" reply. In
this case, "reliable transport" just refers to a lack of content and order errors, not that messages always
end up effective at the remote destination.

140   On CAN links, the datagram protocol is constrained by the need the limited frame size. Datagrams
must be broken into multiple frames for transfer. The protocol must allow for the possibility that
datagrams to a particular node can be interleaved if several nodes are transmitting at the same time.
Note that a node can control what it sends, so it is possible to ensure that only one datagram is being
sent from a node at one time, just not that only one is being received.

145   We're relying on CAN not reordering frames, even in the presence of error recovery. That's needed in
any case for streams, which are so large that complete internal buffering cannot be assumed.

3.3 Examples

3.3.1 Normal CAN case
_____Datagram First Frame →
150   _____Datagram Middle Frame →
_____Datagram Middle Frame →
_____Datagram Last Frame →
_____← Datagram Received OK

155   ### 3.3.2 Short CAN datagram case
_____Datagram Only Frame →
_____← Datagram Received OK

3.3.3 Rejected short CAN datagram case
160   Even a short datagram can be rejected with a "Temporarily unable to receive" if there's e.g. a shortage
of buffers:

_____Datagram Only Frame →
_____← Datagram Rejected, Buffer shortage, Resend

165   ### 3.3.4 Rejected interleaved CAN datagram case:
Accidental interleave, at a node that can't handle that:

_____Datagram First Frame from A →
_____Datagram First Frame from B →
170 _____← Datagram Rejected, Buffer shortage, Resend to B
_____Datagram Middle Frame from A →
_____Datagram Last Frame from A->
_____← Datagram Received OK to A
_____Datagram First Frame from B →
175 _____Datagram Final Frame from B->
_____← Datagram Received OK to B

3.3.5 Command/Reply Example
_____Command Datagram →
_____← Datagram Received OK
180 _____← Reply Datagram
_____Datagram Received OK →

3.3.6 Back and Forth Datagram Protocol

Small nodes may have only one datagram receive buffer and one datagram transmit buffer. In that case, sending multiple commands can result in poor performance if the replies go after the next request, e.g.:

185 _____Command Datagram →
_____← Datagram Received OK
_____← Reply Datagram
_____Command Datagram →
_____Datagram Received OK →
190 _____← Datagram Received OK
_____← Reply Datagram
_____Datagram Received OK →

Note that the 2nd command was sent before the reply datagram was sent. This is valid, but nodes with only one buffer may not be able to handle that Read until the transmit buffer has been emptied by an
195 acknowledgement. In that case, the sequence may become e.g.:

_____Command Datagram →
_____← Datagram Received OK
_____← Reply Datagram
_____Command Datagram →
200 _____← Datagram Rejected (reject, no buffer, please resend)
_____Datagram Received OK →
_____Command Datagram (retransmit) →
_____← Datagram Received OK
_____← Reply Datagram
205 _____Datagram Received OK →

which is significantly less efficient. Better to send the positive acknowledgement to the read reply datagram before sending the next read command.

3.4 Datagram Content

210 There needs to be a method for identifying the content of a general datagram. OpenLCB is meant to be used with both very simple, low-end nodes, and high-end computers with multiple programs. In both those environments, it's inconvenient to have constraints like "the next datagram will carry …." because there might be more than one thing going on, and it's hard to write code that knows about lots of global state information. It must be possible to interleave e.g. configuration datagrams (messages)

215 with datagrams that are displaying messages. Therefore, most datagrams must carry information about which conversation they are part of. We call that "datagram content identification".

We also want those "datagram content IDs" to be small, unique, unambiguous, easily assigned, etc. To do that, there are two basic approaches:

220 • We define a small (one-byte) ID field with a central sequential method of assignment. It's called the "Datagram Protocol ID". This is small, so doesn't use much space on CAN, but requires central recording of IDs.

• We considered using a 6 byte field assigned the same way that node unique identifiers are. The protocol designer(s) uses a unique ID they control to identify the new protocol they are defining. This takes significantly more space, particularly on CAN, but is simple to maintain

225 and requires no central registration.

The plan is to use both. There will be a one-byte field, with certain values to escape to a two-byte field (reserved for future use), and to a six-byte fully-unique field. A separate spreadsheet is accumulating values.

3.5 Numerology

230 This section discusses how big the maximum datagram payload can be. This is a decision that must balance buffer size versus utility of the protocol. Memory is scarce in a small node. Although it's possible to incrementally process parts of a datagram as they arrive, in general it will be necessary to have a buffer of the maximum possible size available when a datagram is arriving. (That same buffer can be used to send datagrams, if needed) At the same time, datagrams that are too small to contain a

235 useful atomic message cause a lot of extra coding work.

If one byte is used for the datagram format field, the first CAN-link datagram contains 7 payload bytes (the following datagrams can contain 8). The datagram format field counts as part of the information that the datagram is transporting, though. We could make the maximum datagram size nine segments, $(9*8) = 72-1$ data bytes, to ensure that even with a little bit of higher protocol, say up to 6 bytes, 64

240 bytes of actual data can be delivered at a shot.

Alternately, because of the binary nature of addressing, a 64-byte buffer size might be more convenient that a 72-byte one, even at the cost of some lost possible capacity in the last frame.

For external protocols, e.g. support for LocoNet, XpressNet, DCC, etc, it's a great simplification if the datagrams can carry any basic message of the external protocol in a single datagram. Is 64 data bytes

245 then enough? NMRA DCC is up to 5 bytes, including addressing, so it's clearly OK even in a 1st CAN frame. Others?

Our notional decision is nine segments, 72 bytes, but developers should code this as a constant to the extent possible.

3.6 Future Extensions

250 In this version of the datagram protocol, there is no provision for multiple recipients of a single message. Transmission is strictly one-to-one. This isn't because of addressing (making them globally visible isn't hard), but rather the need for guaranteed buffering at the receiving node. In the future, we may want to provide one-to-many datagrams.

The recipient needs to have a buffer into which any datagram in flight can be received. On a tiny node,
255 those might be a scarce resource. But, without any protocol support, e.g. if datagrams were "fire and forget" like event notifications, you might need to have a very large number of those buffers. There's nothing in the protocol that prevents nodes A, B, C and D from firing off a datagram at node X at the same time, and having their CAN frames be interleaved when they arrive at X.(The stream protocol negotiates this in advance, but that requires time and resources only appropriate to large transfers) To
260 properly receive them, X needs N=4 buffers to reassemble them in this case. But how big an N is really required? It's not possible to know, so instead we add protocol support: X can tell nodes sending it datagrams to repeat them. That way X can accept as many datagrams as it has buffers (at least one), while telling the others to "say that again, please, I wasn't paying attention the first time".

A sending node knows that it has to hang onto the datagram until it gets a "acknowledged" message,
265 because if it gets a "say that again" it needs to still have the content to resend it. The "acknowledged" and "say that again" are not about data loss on a link, but about buffer management in tiny nodes. A one-to-many protocol could also do something like that, because the transmitter still knows how many "many" is. Global broadcast, like events, is hard because the transmitter's buffer management gets complicated unless it knows how many nodes are listening for it's reply.

270 Bosch is proposing a new standard called CAN-FD with increased bit-rate over the data portion allowing up to 64 bytes! http://www.semiconductors.bosch.de/media/pdf/canliteratur/can_fd.pdf

Decided not to reduce max data length to fit easily within this for ease of migration in the long-term. There are good reasons to retain the possibility of larger-than-64-byte datagrams, e.g. so you can have a header of N bytes on the front of 64 data bytes. All that's needed is to ensure the datagram spec can
275 work with larger frames of any length.

3.7 Implementation Notes

A resource-constrained node can get along with a single datagram buffer for both sending and receiving, but this may require that the node be able to abandon and recreate a datagram that it is trying to transmit. This can happen when e.g.:

280 • Node A creates a datagram for node B in its buffer and starts to send it.

• Before starting to receive the datagram from A, node B creates a datagram for Node A in its buffer and starts to send it.

Both datagrams now need a place to put the incoming datagram, but only have one buffer in this example. They can both send "Datagram rejected; buffer shortage", but then their peer will just repeat
285 the attempt to send. At least one has to drop the content from its datagram buffer and receive the

datagram to avoid deadly embrace. This can also happen in other contexts too, but only when sharing a buffer between transmit and reception. It's therefore easiest for an implementation to provide for two buffers if it's likely to send and receive datagrams asynchronously.

290 The datagram protocol allows all nodes to reject a datagram when it's been totally received. CAN and other wire protocols also allow an early rejection to help keep buffers clear. It's not required that CAN implementations use that, though.

Table of Contents