OpenLCB Technical Note	
OpenLCB-CAN Datagram Transport	
Jul 23, 2011	**Preliminary**

1 Introduction

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

2 Annotations to the Specification

5 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

10

Below this is just the content of the Datagram development doc right now.

Some wire protocols can support only very short packets/frames, e.g. CAN with a limited header and data payload. As OpenLCB evolves, it will inevitably need messages larger than that.
15 Streaming provides a very large payload, at some cost in setup time and complexity. "Datagram Support" is in between "small enough to always be atomic" and "large enough to justify streaming overhead".

This note is a working draft proposal.

3 Use Cases

20 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
25 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 30 the future.

4 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 35 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 40 a balance between these, and to optimize certain layouts on CAN links, a size of 72 bytes has been 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. 45 CAN, TCP/IP, etc, handle their own error conditions. The datagram protocol needs flow control and 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 50 end up effective at the remote destination.

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 55 sent from a node at one time, just not that only one is being received.

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.

5 Base Protocol Proposal

In the base protocol a datagram is just a single short message containing the Datagram MTI and the 60 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 Acknowledged" 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 protocol.

65 Instead of "Datagram Acknowledged", the receiving node may return a "Datagram Rejected" message tagged to represent exactly one of several conditions:

"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:

70 "Information Logged" - the node supports the logging protocol and information was logged for later retrieval.

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

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

75 "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.

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

6 CAN protocol proposal

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

85 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.

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. Two format indicators are used to mark
90 non-final and final frames. The data part of the CAN frame can carry from zero to eight data bytes.

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 last frame is marked by a "datagram complete" format indicator.

The receiving node receives the frames into a buffer. The frames from more than one datagram may
95 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. If sufficient buffers are not available, the receiving node rejects the datagram and it will be resent. 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.

100   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.

6.1 CAN Examples

6.1.1 Normal CAN case

105   Datagram Segment →

Datagram Segment →

Datagram Segment →

Datagram Segment (Complete) →

110   ← Datagram Acknowledged

6.1.2 Short CAN datagram case

Datagram Segment (Complete) →

← Datagram Acknowledged

115   ### 6.1.3 Rejected short CAN datagram case

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

Datagram Segment (Complete) →

120   ← Datagram Rejected, Buffer shortage, Resend

6.1.4 Rejected interleaved CAN datagram case:

Accidental interleave, at a node that can't handle that:

125   Datagram Segment from A →

Datagram Segment from B →

← Datagram Rejected, Buffer shortage, Resend to B

Datagram Segment from A →

Datagram Segment (Complete) from A->

130   ← Datagram Acknowledged to A

Datagram Segment from B →

Datagram Segment (Complete) from B->

← Datagram Acknowledged to B

7 Datagram Content

135   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)
140   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:

Have a small (one-byte) ID field, and have some central sequential method of assignment. This is
145   small, so doesn't use much space on CAN, but requires central recording of IDs.

Use 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 and requires no central registration.

The plan is to use both. There will be a one-byte field which can be combined with flags on CAN, with
150   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. (spreadsheet) (pdf)

8 Numerology

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
155   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 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
160   (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 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
165   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 then enough? NMRA DCC is up to 5 bytes, including addressing, so it's clearly OK even in a 1st CAN frame. Others?

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

9 Extensions

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

175 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, 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

180 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 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

185 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, 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

190 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.

10 Implementation Notes

A resource-constrained node can get along with a single datagram buffer for both sending and

195 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.

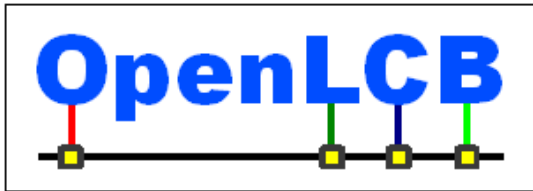Node A creates a datagram for node B in its buffer and starts to send it.

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

200 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 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

205 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.

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.

210

Table of Contents

10