OpenLCB Technical Note	
OpenLCB-CAN Datagram Transport	
Jan 28, 2013	Draft

1 Introduction

This Technical Note contains informative discussion and background for the corresponding "OpenLCB Datagram Transport Standard". 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 Standard

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

2.1 Introduction

2.2 Intended Use

2.3 References and Context

For more information on format and presentation, see:

- OpenLCB Common Information Technical Note

2.4 Message Formats

2.4.1 Datagram Content

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

Note that zero-length datagrams are valid.

2.4.2 Datagram Received OK

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

2.4.3 Datagram Rejected

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

Rejecting datagrams is meant for cases where the actual transmission and reception of the datagram has failed. It's not meant for higher level errors, such as “The command to move the points did not succeed because they were stuck”. The decision whether to reply with Datagram Received OK or Datagram Rejected needs to be made promptly, so that the reply can be sent, buffers can be cleared and communications can continue.

2.4.3.1 Error Codes

In general, errors are of one of two types:

- “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 particular datagram. The datagram should not be retransmitted.
- “Resend OK” — In contrast to the error above, the problem accepting the datagram is of a transient nature, and the sending node should attempt retransmission at its discretion at some point in the immediate to near future.

The standard is written to allow for the case where neither is set if the node doesn't have a clear idea how to handle the error, but this is an undesirable case.

Optionally, the node can mark the reply with one or more of several conditions:

“Source not permitted” — Datagrams from this source will never be accepted. Generally, this flag is only meaningful in conjunction with the “Permanent Error” flag, as there is no sense in retransmission in this context. This flag might be used to indicate that a receiving node is only willing to establish a connection with an exclusive set of nodes that doesn't include the sender.

“Datagrams not accepted” — This node will not accept datagrams of this type under any circumstance. If the node implements the Datagram Transport protocol, it should use this to reject datagrams that it cannot process because they are of an unknown type. Rejecting the interaction with an Optional Interaction Rejected message (see “OpenLCB Message Network Standard”) should only be used by nodes that don't implement the Datagram Transport protocol because it carries the notion that the Datagram Protocol isn't supported.

“Transport Error” — Something appears to have gone wrong between the sender and the recipient, perhaps because the recipient received datagram fragments out of order, or a sequence of datagrams was received out of order (at the application layer). This error flag does not by itself imply that the sender should retransmit; rather, this flag is be combined with the “Resend OK” flag to explicitly request retransmission.

"Buffer shortage" - The node wasn't able to receive the datagram because of a shortage of buffers. This can happen during normal operation due to overlapping traffic to a node. Setting this bit helps diagnose when datagram traffic is overwhelming the node's ability to buffer and process it

"Out of Order" - Should not happen, but an internal inconsistency was found in the OpenLCB message sequence and/or in the CAN frames making up a datagram. This is probably a subset of the Transport error cases.

The least-significant bits can be used for code for the specific condition that lead the receiving node to reject the Datagram Content message. This is a good idea, because it helps debug persistent problems. Node manufacturers should make the definition of those codes available in documentation, but that's not required by the Standard.

2.5 States

A datagram receive buffer is a form of state information, in that it's either full with a message being processed, or empty and ready to accept a new datagram. The protocols are written so that implementations can use just one buffer, or many. Depending on the connection to the physical OpenLCB network being used, it may be possible for overlapping reception of datagrams. This works even with one buffer, because datagrams that arrive while the first is still in process can be rejected temporarily. More buffers can make that more efficient.

2.6 Interactions

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 for use with application layer protocols transported over datagrams, see §3.4 below.

2.6.1 Normal Transmission

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 are required to be signaled using another protocol.

2.6.2 Rejected Transmission

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).

There's no requirement for a specific response to out-of-spec Datagram Content messages. If a node receives out-of-spec datagram messages, such as one with too many bytes, what it does is up to the receiving node. This is because it is 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.

2.7 Implementation via CAN Transport

2.7.1 CAN Message Formats

For CAN, the Datagram Content 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.

- CAN-Datagram Content Single Frame — Note that zero-length datagrams are valid.
- CAN-Datagram Content First Frame — Note that there's no requirement that this contain 8 bytes. This is to keep the total number of frames in a Datagram Content message bounded.
- CAN-Datagram Content Middle Frame — Note that there's no requirement that this contain 8 bytes. It cannot be empty, though.
- CAN Datagram Content Last Frame — Note that this frame can be zero length.

2.7.2 CAN States

The CAN implementation allows more than one datagram to arrive at a node at the same time, since datagram fragments from node A and node B can arrive interleaved at their destination node C. Datagram implementations need to track this in their internal state. Approaches include:

- Having a single buffer, in which the fragments that go with the first-received Datagram Content First Frame are accumulated. Other datagrams can then be rejected immediately when their Datagram Content First Frame or Datagram Content Single Frame arrive.
- Having multiple buffers in which the datagrams are separately accumulated.

Any implementation that obeys the protocol constraints in the Standard is acceptable.

2.7.3 CAN Interactions

2.7.3.1 Normal Transmission

The sender breaks the datagram content into one or more frames. The source NodeID alias and destination NodeID alias are, unlike other addressed messages, both contained in the header. 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.

If there are more than eight bytes to the datagram, the rest are sent 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 in the CAN header to aid in reconstruction of the full datagram in the buffer.

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 dynamic content in the source node.

It is preferred that the frames making up a datagram be transmitted in a group, adjacent on the CAN bus. It is permitted to interrupt that to send higher-priority frames, either for CAN segment-level control, or for higher priority messages. It is not permitted to send lower priority frames, nor to interleave multiple datagrams, so that the buffering requirements at the receiving nodes stay bounded.

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 to a single destination node.) If sufficient buffers are not available, the receiving node rejects the second datagram with a "Resend OK"-flagged error code. The reject message 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 (i.e., an Only datagram or Last datagram message) 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 the node Initialized state is reset so 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.

2.7.3.2 Rejected Transmission

There's no requirement for a specific response to out-of-spec transmissions. If a node receives out-of-spec datagram fragments, what it does is up to the receiving node. This is because it is 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.

For example, a node that receives a Middle datagram without a First datagram can react in various ways. It might send "Resend OK", with or without "Transport error" and/or "Out of order", or simply ignore the message entirely.

Notice, however, that receiving a second valid datagram sequence from yet another node 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 in this case. The receiving node is required to send a Datagram Rejected message usually (but not necessarily) with the "Resend OK" error code.

3 Background Information

3.1 Use Cases

Here are some example application-level protocols that are or might be transported using datagrams.

- Memory 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 an OpenLCB segment 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. Communication 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 by the datagram transport.

- RFID: RFID tags carry 40-64 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 single 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 is not possible to guarantee in advance that a buffer is available in the destination to receive and assemble the datagram.

The datagram protocol does not require error correction, because OpenLCB is based only on reliable links, that is, links that guarantee no content or order errors. CAN, TCP/IP, etc., handle their own error correction. 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. The “Received OK” message is to inform small-capacity sending nodes that it can free its buffer holding the transmitted message.

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.

The datagram content formats rely 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 →
Datagram Middle Frame →
Datagram Middle Frame →
Datagram Last Frame →
← Datagram Received OK

3.3.2 Short CAN datagram case

Datagram Only Frame →
← Datagram Received OK

3.3.3 Rejected short CAN datagram case

Even a short datagram can be rejected with a “Resend OK” if there's, e.g. a shortage of buffers:

Datagram Only Frame →
← Datagram Rejected, Resend OK

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 →

← Datagram Rejected, Resend OK to B
Datagram Middle Frame from A →
Datagram Last Frame from A →
← Datagram Received OK to A
Datagram First Frame from B →
Datagram Final Frame from B →
← Datagram Received OK to B

3.3.5 Command/Reply Example

Command Datagram →
← Datagram Received OK
← 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.:

Command Datagram →
← Datagram Received OK
← Reply Datagram
Command Datagram →
Datagram Received OK →
← 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 acknowledgement. In that case, the sequence may become, e.g.:

Command Datagram →
← Datagram Received OK
← Reply Datagram
Command Datagram →
← Datagram Rejected, Resend OK
Datagram Received OK →
Command Datagram (retransmit) →
← Datagram Received OK
← Reply Datagram
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

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 is inconvenient to have constraints like “the next datagram will carry…” because there might be more than one thing going on, and it is hard to write code that knows about lots of global state information. It must be possible to interleave, e.g., configuration datagrams (messages) with datagrams that are displaying messages. Therefore, most datagrams will carry information about which conversation they are part of. We call that information “datagram content identification”.

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

- A one-byte ID field with a central sequential method of assignment. It is called the “Datagram Protocol ID”. This is small, so doesn't use much space on CAN, but requires central recording of IDs.
- 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, 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

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 is 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, requiring extra coding work.

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.

At the low end, datagrams should go all the way down to zero bytes, because they are distinct from Events (which have a fixed 8-byte payload).

If one byte is used for the datagram content ID field (see §3.4), 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 additional overhead from an application-layer 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 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 is 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? The evidence at hand suggests that 64 data bytes is more than sufficient.

As a balance between these, and to optimize certain layouts on CAN links, a size of 72 bytes has been selected. The OpenLCB protocols have been designed such that this can be extended to 128 bytes for specific application protocols if needed, while still maintaining backward compatibility with devices that only implement a 72-byte datagram size.

3.6 Future 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 constraint is not because of addressed nature of datagrams (making the messages globally visible is not hard), but rather the need for guaranteed buffering at the receiving node. In the future, we may want to provide one-to-many datagrams.

For this to work, the recipient will need 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 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? The recipient cannot 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 receives a “Received OK” message, because if the sender receives a “Rejected, Resend OK” it needs to still have the content to resend it. The “Received OK” and “Rejected, Resend OK” 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 its reply.

Although Bosch has proposed a new standard called CAN-FD[1], which permits frames as large as 64 bytes, we have decided not to reduce maximum size of a datagram to fit easily within this frame size. 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. Should CAN-FD come into general use, datagrams will just use the longer frames in the current system for breaking Datagram Content messages into segments.

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.:

[1] http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd.pdf

- 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 “Rejected, Resend OK”, 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 buffer between transmit and reception. It is therefore easiest for an implementation to provide for two buffers if it is likely to send and receive datagrams asynchronously.

The datagram protocol allows all nodes to reject a datagram when it has been totally received. CAN and other wire protocols, because they require fragmentation of the datagram into smaller segments, therefore also allow an early rejection of a datagram. This mechanism can help keep buffers clear, but it is not required that CAN implementations use this mechanism.

Table of Contents

