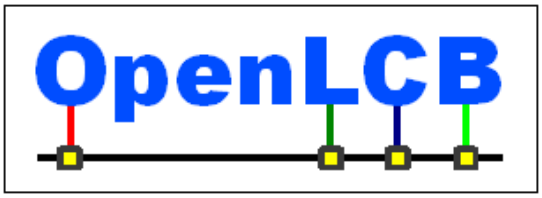


OpenLCB Technical Note	
Memory Access Configuration Protocol	
Oct 18, 2013	Preliminary

1 Introduction

This Technical Note provides background for the association Memory Access Configuration Protocol Standard.

The protocol is called “memory access” because it makes the configuration information in the node look like it's stored in linear memory spaces.

- Don't assume that the information actually has to be stored that way, as linear memory. The node can remap the information that's being read or written into whatever internal organization it needs.
- Don't assume that no other configuration protocol exists. Some day, for example, there might be a “file access configuration protocol” where a node pulls predefined configuration information from some central store of “files”. Or something else. There's already the teach/learn method of configuring events.
- Don't assume that this is only used for persistent configuration information. The protocol is already being used for e.g. retrieving the CDI, and for providing a simple debug capability that allows programers to read (and even write) raw node RAM. It can be used for other things in the future using the memory space mechanism.

2 Annotations to the Standard

2.1 Introduction

Note that this section of the Standard is informative, not normative.

2.2 Intended Use

Note that this section of the Standard is informative, not normative.

2.3 Reference and Context

For more information on format and presentation, see:

- OpenLCB Common Information Technical Note

2.4 Protocol

2.4.1 Address Space Size

The four-byte address allows directly addressing 4GB of data. The use of address spaces (see below) allows direct access to 1TBi.

The large address range removes the need for address registers and other non-idempotent accesses when accessing e.g. sound information in a large memory.

2.4.2 Address Space Selection

Although a 32-bit address space is large enough to cover combined uses of memory, it can be more convenient to consider separate address spaces in the node. (This can also be considered to be a top digit in a global address space, if you want to, but note that the separate address spaces may cover the same memory objects, e.g. “all memory” and “Event ID configuration” spaces may reference the same configuration memory)

2.4.3 Message Formats

This section is about helping implementors see how to decode the message format. It's not normative; if this section differs from the Standard, the Standard rules.

Configuration messages use a specific datagram format consisting of the datagram type byte, followed by a single byte combining the operation “Command Type” field and flags. This is then followed by data in an operation-specific format.

Byte 1 has a number of decodable fields. The top two bits of byte 1 separate the basic types:

- 0b00 – Write operations
- 0b01 – Read operations
- 0b10 – Control operations
- 0b11 – Not used / reserved

		Write	Write Under Mask	Write Stream	Write Reply
Datagram Type Byte		0x20	0x20	0x20	0x20
Command Byte (Upper is MSB, lower is LSB)	Command Type (2 bits)	0b00	0b00	0b00	0b00
	Stream/Datagram (1 bit)	0b0	0b0	0b1	0b0
	Reply (1 bit)	0b0	0b0	0b0	0b1
	Under Mask (1 bit)	0b0	0b1	0b0	
	Fail/OK				0b1 Fail 0b0 OK
	Reserved (1 bit)	0b0	0b0	0b0	0b0
	Address Space (2 bits)	0b00-0b11	0b00-0b11	0b00-0b11	0b00-0b11
	Full byte value	0x00-0x03	0x18-0x1B	0x20-0x23	0x18 – 0x1B Fail 0x10 – 0x13 OK
4 bytes		Address	Address	Address	Address
Optional byte, present when Address Space bits above == 0b00		Address Space	Address Space	Address Space	Address Space
Remainder		Data (1-64 bytes)	Data and Mask (2-64 bytes)	Stream Definition Info	Count (1 byte, 1-64)

		Read	**Read-Reply**	**Read Stream**
Datagram Type Byte		0x20	0x20	0x20
Command Byte (Upper is MSB, lower is LSB)	Command Type (2 bits)	0b01	0b01	0b01
	Stream/Datagram (1 bit)	0b0	0b0	0b1
	ReadReply/Write (1 bit)	0b0	0b1	0b0
	Fail/OK		0b1 Fail 0b0 OK	
	Reserved (1 bit)	0b0		0b0
	Reserved (1 bit)	0b0	0b0	0b0
	Address Space (2 bits)	0b00-0b11	0b00-0b11	0b00-0b11
	Full byte value	0x40-0x43	0x58 – 0x5B Fail 0x50 – 0x53 OK	0x70-0x73
4 bytes		Address	Address	Address
Optional byte, present when Address Space bits above == 0b00		Address Space	Address Space	Address Space
Remainder		Count (1 byte, 1-64)	Data (1-64 bytes)	Count (4 bytes), then Stream Definition Info

		Get Config Options	**Get Config Options Reply**	**Get Addr Space Info**	**Get Address Space Info Reply**	**Freeze/ Unfreeze**
Datagram Type Byte		0x20	0x20	0x20	0x20	0x20
Command Byte (Upper is MSB, lower is LSB)	Command Type	0b10	0b10	0b10	0b10	0b10
	Operation Type	0b0000	0b0000	0b0001	0b0001	0b1000
	Reply	0b0	0b1	0b0	0b1	
	Reserved					0b0
	Freeze/ Unfreeze					Freeze 0b1 Unfreeze 0b0
	Reserved	0b0	0b0	0b0	0b0	
	Byte value	0x80	0x82	0x84	0x86	0xA0-0xA1
Byte 2			Available Commands (2 bytes)	Address Space	Address Space	
Byte 3					Largest Address (4 bytes)	
Byte 4			Write Lengths			
Byte 5			Highest Address Space			
Byte 6			Lowest Address Space (0-1 byte)			
Remainder			Name (0-N bytes, optional		Requires Alignment (4 bits) Low Address Non-zero (1 bit) Read-Only (1 bit)	
					{Lowest Address } (4 bytes)	
					{Desc } (0-N bytes)	

		Lock/ Reserve	Lock/ Reserve Reply	Get Unique ID	Get Unique ID Reply	Indicate	Update Complete	Reset/ Reboot	Reinit/ Factory Reset
Datagram Type Byte		0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20
Command Byte (Upper is MSB, lower is LSB)	Command Type	0b10	0b10	0b10	0b10	0b10	0b10	0b10	0b10
	Operation Type	0b0010	0b0010	0b0011	0b0011	0b1001	0b1010	0b1010	0b1010
	Reply	0b0	0b1	0b0	0b1				
	Reserved	0b0	0b0	0b0	0b0				
	Start/Stop					Start 0b1 Stop 0b0			
	Sub Command						0b00	0b01	0b10
	Byte value	0x88	0x8A	0x8C	0x8D	0xA2-0xA3	0xA8	0xA9	0xAA
Byte 2		Requesting NodeID	Locking NodeID	Number to reserve (3 bits, 1-8)	New Unique EventID (8 bytes, 1-8 times)				Target NodeID
Byte 3									
Byte 4									
Byte 5									
Byte 6									
Byte 7									
Remainder									

2.4.4 Operations

This protocol consists of a idempotent operations. A command are sent and cause some desired thing to happen. That in turn results in a positive reply, the return of the requested information, or an error code. Each operation is independent of others and carries all the information needed to do the operation. For example, a read operation contains the address information, and the reply carries that information back to the requestor. This way, no state information needs to be maintained at either end.

Read and write operations to memory should take place very quickly, so their replies (see below) should be close to immediate. But it's also possible for them to take longer, for example to write some

media, or read a DCC CV. The Standard is silent on how long such operations should take. It does provide a way for the node handling the request to indicate the expected time back to the requesting node, by including that in the Datagram Received OK reply message. But that's not required, and the reply can just say “unknown time”.

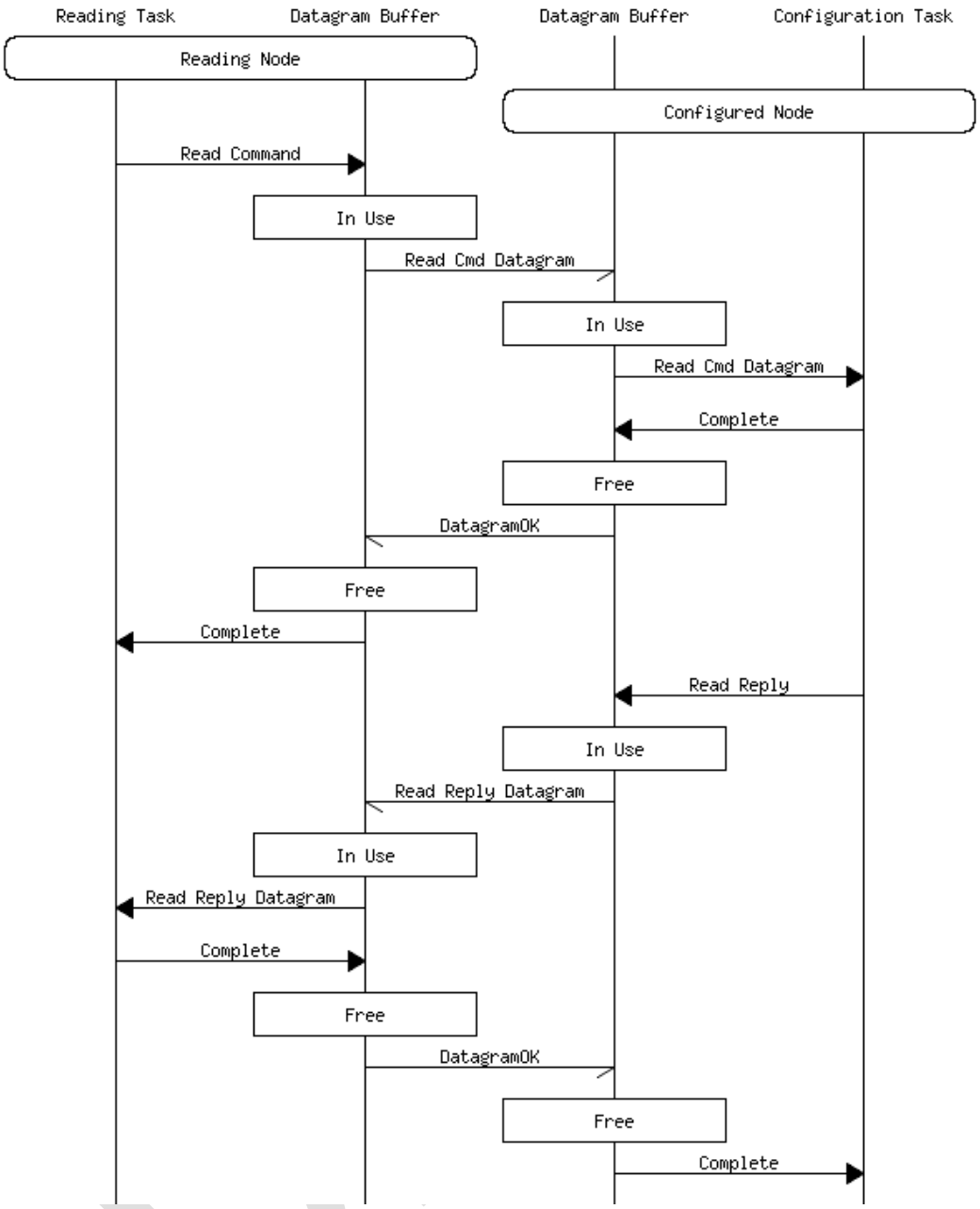
The Standard doesn't limit how messages can be interleaved. The simplest method is to send one command, wait for the reply sequence to be complete, and send the next. Similarly, the simplest implementation for the request-handler node is to accept one command, and work on it until it's done; if another comes along in the meantime, reject the datagram and ask that it be resent until later.

It's also possible to create more performant nodes. A request-handler node could accept more than one request, queue them up, and reply to them when done. Alternately, a node that does very slow operations, like reading CVs from a DCC locomotive in response to read operations, may terminate its current read when it gets another operation. This would allow the user to say “I don't want to wait any more, let's move on” during a long operation.

2.4.4.1 Read, Read-Reply

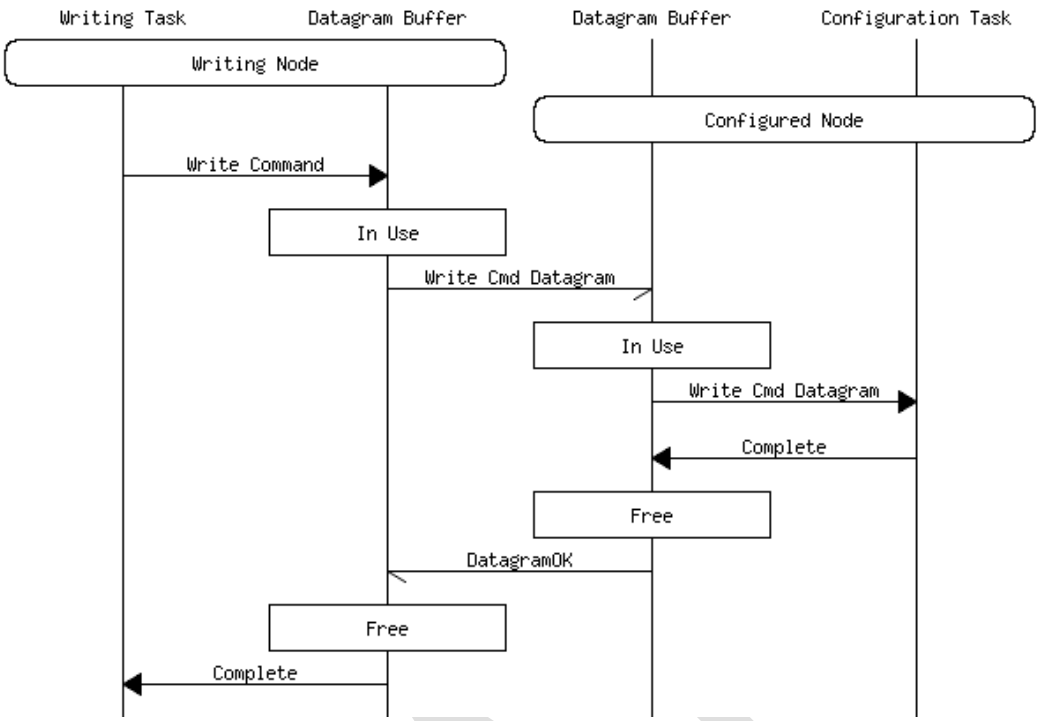
The figure shows a typical read operation. The Read Command is carried to the device being configured by a datagram, the read operation takes place, and the results are returned in another datagram.

If the memory configuration protocol is being used to control e.g. DCC CV reads and writes, those operations can take a very long time, 30 seconds or more.



For reads, the node can return information in the Datagram reply for the Read command that indicates how long it's likely to be before the read is complete and the Ready Reply is returned. That in turn can, but doesn't have to be, used to indicate in-progress status.

2.4.4.2 Write, Write-Reply

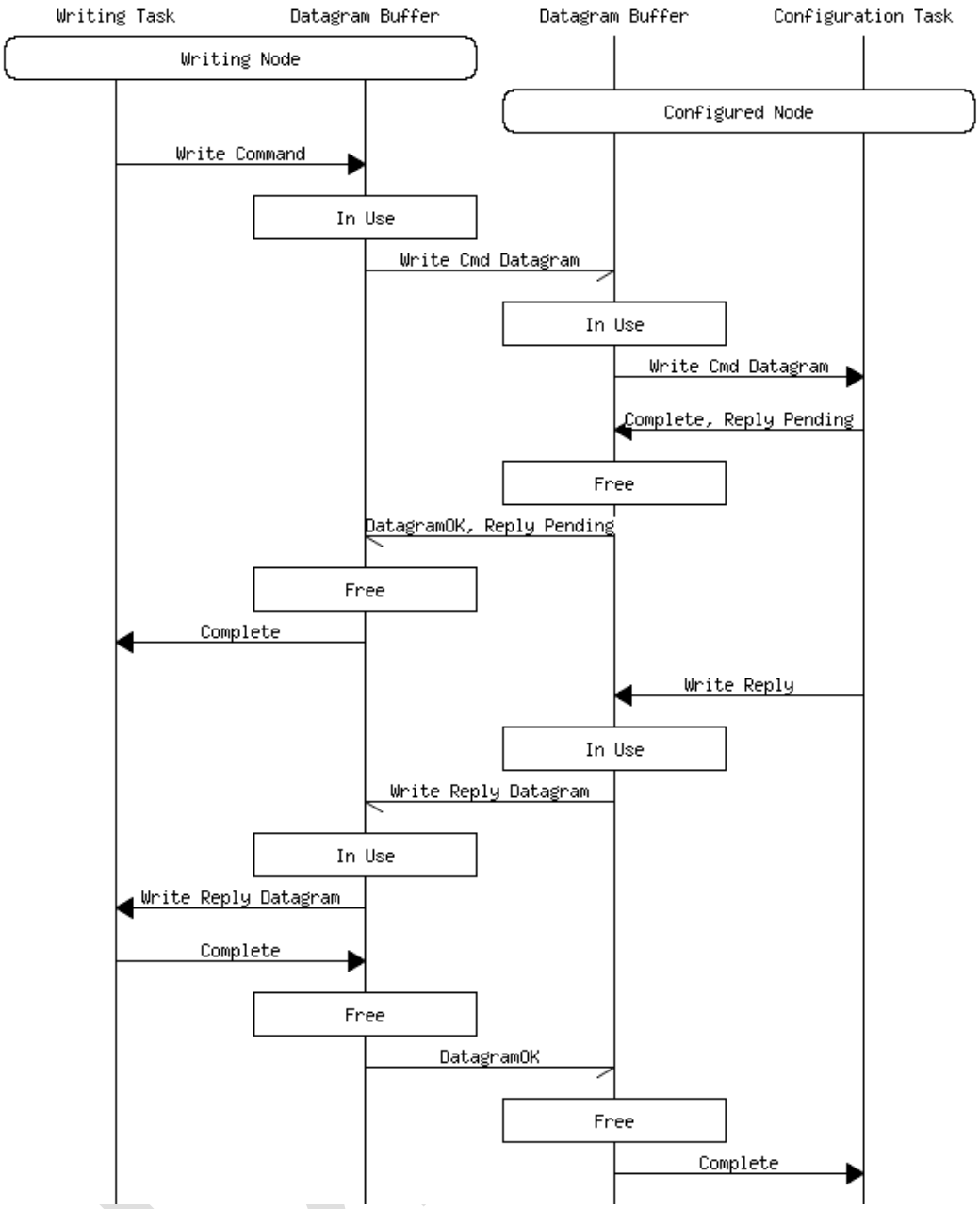


A typical memory write operation. The Write Command and data are carried to the node being configured in a single datagram. Since the write operation succeeds immediately, the only reply needed is the datagram reply.

If the memory configuration protocol is being used to control e.g. DCC CV reads and writes, those operations can take a very long time, 30 seconds or more.

Most writes are assumed to always complete immediately and without error. Writes into memory are certainly like that, but writes to DCC CVs might not be. Into-memory writes are well served by "when the positive reply to the datagram arrives, the write completed successfully". That requires no extra traffic to handle other cases.

For the less-common case of writes that might fail and/or take a long time, the reply to the Write datagram can carry a "will reply later" indication, followed by a Write Reply when the operation is done. That Write Reply, in turn, informs the requestor when the operation was complete, and whether it completed OK.



100 **A delayed response, either because the write takes significant time or could fail for some reason. The datagram reply to the Write Command carries the reply pending bit set, and a reply datagram is send from the configured node back to the configuring node with the reply.**

2.4.4.3 Get Configuration Options Reply

2.4.4.4 Get Address Space Information Reply

2.4.4.5 Lock/Reserve and Freeze/Unfreeze

2.4.4.6 Get Unique EventID

Nodes maintain a list of unique EventIDs for use in configuration. These are allocated based on the node's unique NodeID. This command allows a configuration tool to get new unique EventIDs from the node's pool, for example to interact with the Blue/Gold configuration process. Each request must provide a different EventID, without repeat, even through node resets and factory resets.

2.4.4.7 Update Complete/Reset/Reboot/Reinitialize

2.4.4.8 Indicate

This is likely to move to another standard.

3 General

3.1 Environment of Protocol

3.1.1 Requirements

- Nodes must carry enough context that a stand-alone configuration tool can provide a useful human interface without getting any data from an external source, e.g. needing an Internet download to handle a new node type.
- It must be possible to configure a node entirely over the OpenLCB, without physical interactions, e.g. pushing buttons. This configuration must be compatible with local configuration, including e.g. the Blue/Gold method.
- It must be possible to configure one or more nodes while the rest of the OpenLCB is operating normally.
- It must be possible to read, store, and reload the configuration of a node.

3.1.2 Preferences

- Small nodes shouldn't need a lot of processing power, e.g. to compress or decompress data in real time. Memory usage should also be limited, but is a second priority.
- Configuration operations should be state-less and idempotent to simplify software at both ends.
- Multiple independent configuration operations can proceed at the same time. Multiple devices should be able to configure separate nodes at the same time. Multiple overlapping reads of the same node should be possible. There should be a method to coordinate separate configuration operations to simplify configuration software.

- For efficiency, atomic reads or writes of small amounts of data should fit into a single-frame CAN datagram. Single-bit writes also add efficiency.
- Multiple address spaces make it easier to handle multiple types of data.
- For large transfers, it's desirable to be able to use streams. Not all nodes support them, though, so it must be possible to enquire about capabilities.
- Addresses should be four bytes. Two address bytes is a failure of imagination.

3.1.3 Design Points

Basic configuration is done with datagrams, which can carry 64 bytes of data to read or write.

For efficiency, writes don't need any reply except the datagram response. By sending that only after the write is complete, including non-volatile memory delays, the written node can control the transfer rate.

Read operations must return data in a separate datagram. If this carries the address, etc, in addition to the data, the operation is idempotent.

Read and write operations can address separate kinds of information in the node via specifying specific address spaces. Three of these have specified uses, and the reset are optional tools for future expansion.

There are a large number of possible configuration options, and more may be developed in the future: Reset, identify, etc. Not all nodes will implement all methods, so a query operation is needed so that tools can know what they can do.

Can't require that nodes do anything in particular to put changes into effect. Some will do it right away, some will require a reset, etc. Need flexibility for node developer, yet consistency for configuration tool builder.

3.1.4 CAN Aspects

CAN configuration datagrams have seven payload bytes in the 1st frame (the first byte is the datagram type byte that identifies that it's a configuration transfer). See the "Datagram Protocol" document for more info. Read operations and 2-byte (or 1-byte-under-mask) writes can be done with a single CAN frame.

Read-Stream and Write-Stream need to rendezvous on the stream IDs to know which stream is carrying the data. See Examples section below.

3.2 Implementation Notes

This section is non-normative notes and suggestions for implementors.

3.2.1 Example CAN Operations

These are over CAN and includes some information from the datagram protocol to make traffic clearer. [d] is a single datagram; [di][di][de] is a datagram that's in multiple frames.

3.2.1.1 Get initial information

Get Configuration Info [d] →

← datagram reply

← Get Configuration Info Reply [d]

datagram reply →

3.2.1.2 Write byte

Write [d] →

← datagram reply

3.2.1.3 Write 64 bytes

Write [di] →

Write [di] (7 times) →

Write [de] →

← datagram reply

3.2.1.4 Read byte

Read [d] →

← datagram reply

← Read Reply [d]

datagram reply →

3.2.1.5 Read 64 bytes

Read [d] →

← datagram reply

← Read Reply [di]

← Read Reply [di] (6 times)

← Read Reply [de]

datagram reply →

3.2.1.6 Large read via stream

On CAN:

Read [di] →

Read [de] →

← datagram reply

← Read Reply [d] (carries stream ID)

datagram reply →

← Stream Initiate Request (carries a buffer size equal to memory write line size)(stream ID from above)

→ Stream Initiate Reply

← Stream Data Send (N times; broken down to frames, 8 bytes each)

Stream Data Proceed →

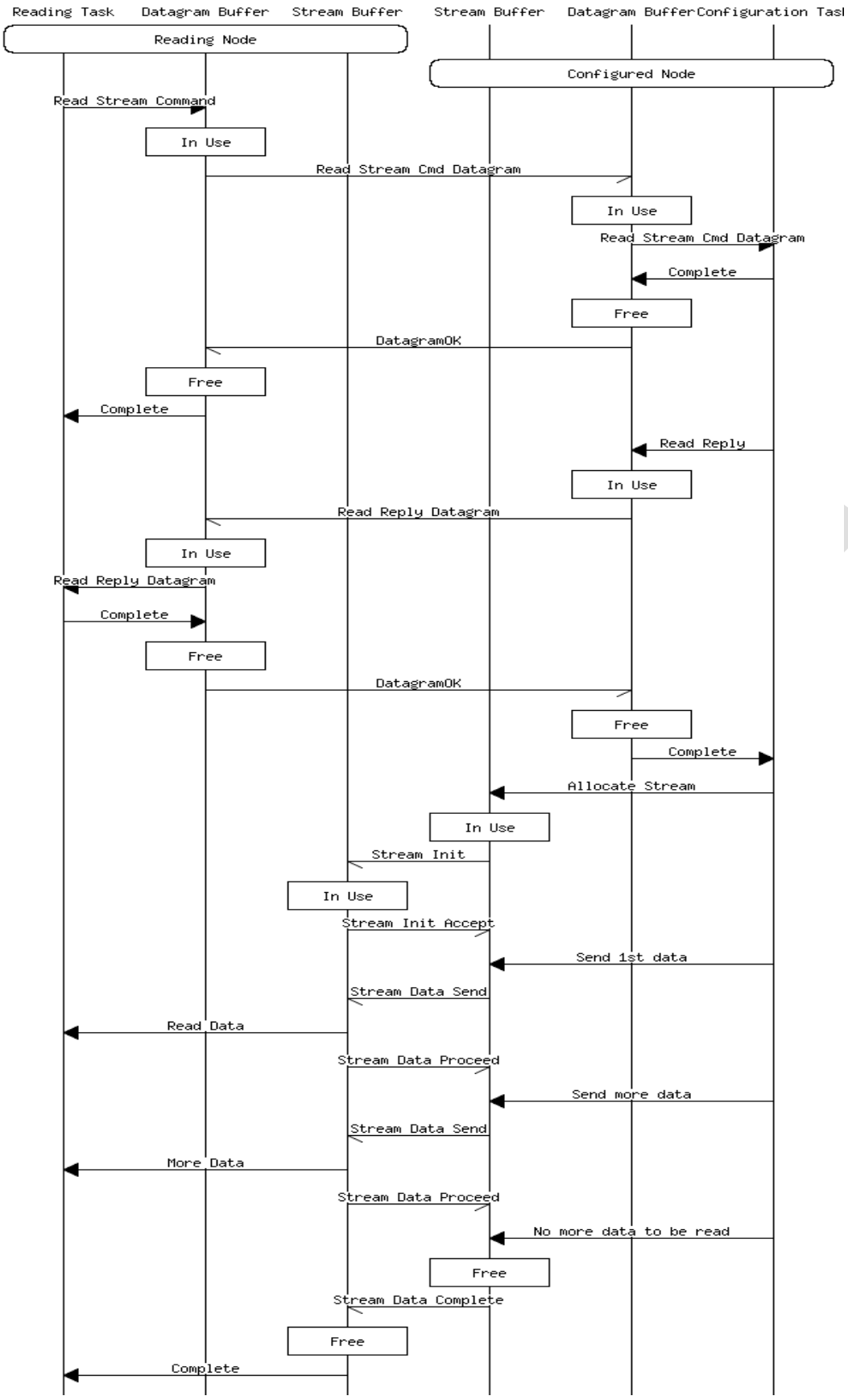
(repeats until done, then)

← Stream Data Complete

First a datagram comes back with reply information, then the stream is created based on the StreamIDs in the messages.

Streams require allocation, which might fail.

- If the stream can't be allocated locally, one option is to reject the incoming Read Stream Command datagram with “try again later”.
- Another option if the stream can't be allocated locally is to send a Read Reply with the Failed bit set. (Need to define error code bits!)
- If the stream fails allocation at the requesting end, the failure of the stream allocation indicates the error, and the requesting node can retry the Read Stream Command.
- If the stream allocation fails between the two nodes, e.g. as it passes through a gateway, the initiating node will know but the requesting node won't hear about it. That should never happen, but how is a failure like that indicated to the requesting node?



3.2.1.7 Large write via stream

Write [d] →

← datagram reply

Stream Initiate Request →

← Stream Initiate Reply

Stream Data Send (N times; broken down to frames, 8 bytes each) →

← Stream Data Proceed (after write to memory complete)

(repeats until done, then)

Stream Data Complete →

Not clear how there can be a write reply in this, e.g. saying "No!". Is that part of the datagram reply? Cover the case to make it definite.

3.2.1.8 Performance Note

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

Read [d] →

← datagram reply

← Read Reply [d]

Read [d] →

datagram reply →

← datagram reply

- ← Read Reply [d]

datagram reply →

Note that the 2nd read command was sent before the reply to the Read-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.:

Read [d] →

← datagram reply

← Read Reply [d]

Read [d] →

← datagram negative reply (reject, no buffer, please resend)

datagram reply →

Read [d] (retransmit) →

← datagram reply

← Read Reply [d]

datagram reply →

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

3.2.2 Delays Due to Non-Volatile Memory

Some microcontrollers can't continue to operate while writing configuration information to non-volatile memory.

If the CAN buffering is sufficient (at about 1usec per buffer) for the node to become active at the end of the memory operation and process buffered frames at the full rate, there's no issue.

If a node has missed one or more frames, it's possible that some state interaction has started and the node is inconsistent. For example, a RIM/CIM sequence could have started or even finished.

If the node misses one or more frames, the CAN controller needs to flag that and bring it to the attention of the node's microcontroller. The node then needs to emit an “Initialization Complete” message so that other nodes realize that this node may not have complete state information.

The node should also defer the reply to a write datagram until after the write is complete, to make it less likely the next-in-sequence operation arrives during dead time. Nodes doing the configuring should limit the amount of traffic they send to the node-under-configuration to reduce the need for e.g. datagram retransmission.

There are bits that allow the node to specify what size writes are allowed. These can be used to force the configuring node to do operations in the most efficient way, e.g. to minimize dead time during writing.

The transfer buffer size for stream access is negotiated. By requiring a transfer size equal to or smaller than the memory write size (page size), the node can ensure that the stream will pause during a write operation.

3.2.3 Large Volume Operations

The stream protocol is meant for large reads and writes, but the datagram protocol can also work well on a single CAN segment. The difference in performance comes from the (potential) larger datagram buffer size.

All nodes support datagrams for configuration; not all support streams. So a least-common-denominator configuration tool would use sequences of datagrams for even large transfers. Because the need for reply, short datagrams are not particularly efficient. In the limiting case, you can only write two bytes per frame exchange. The sending node should look at the Get Configuration reply and use the largest available size.

On the other hand, if non-volatile memory timing requires that write operations to a node use a 64-byte or smaller stream buffer size, then datagrams are a more efficient method than streams. In that case, the node should indicate that stream-write operations are not supported in its reply to Get Configuration. Since read operations don't have the same timing issues, streams may still be useful in that case.

4 To Be Merged

Various OpenLCB capabilities and protocols will define memory spaces starting at the highest numbers. Other uses should start with the low numbered spaces, and should not use spaces with the highest bit set, although that's not a standards-level requirement.

Reminder: The P/C protocol requires that an IdentifyProducers or IdentifyConsumers message be emitted when a configuration change results in the node producing or consuming additional event IDs. This can be done automatically if the configuration change only takes effect when the node is reset. If the change takes effect immediately, the node should emit the necessary messages immediately. It's not necessary for the node to check whether the new Event ID is already produced or consumed; redundant Identify messages are not a problem.

There are some error bits defined in the message tables here that aren't in the Standard. And there are some undefined error bits for which homes need to be defined (from JimK):

- CONFIG_MEM_RESULT_OK : TConfigMemErrorCode = ($00, $00);
- CONFIG_MEM_RESULT_TERMINATE : TConfigMemErrorCode = ($10, $00);
- CONFIG_MEM_RESULT_REJECTED_STREAM_PERMANENT_ERROR : TConfigMemErrorCode = ($10, $10); // $10xx high bytes says don't try again
- CONFIG_MEM_RESULT_REJECTED_STREAM_BUFFER_FULL : TConfigMemErrorCode = ($20, $00); // $20xx high byte says to try again

Table of Contents

