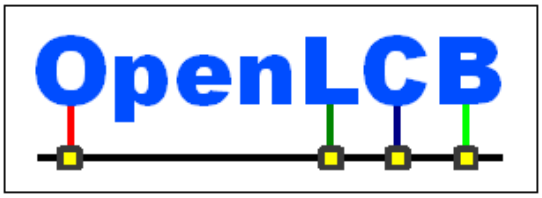

OpenLCB Technical Note	
Memory Access Configuration Protocol	
Mar 27, 2012	Preliminary

1 Introduction

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

3 Stuff to be merged into the above

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.

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.

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.

3.1.2 CAN Aspects

CAN configuration datagrams have seven payload bytes in the 1st frame (the first byte 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.
60 [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]

65 datagram reply →

3.2.1.2 Write byte

Write [d] →

← datagram reply

3.2.1.3 Write 64 bytes

70 Write [di] →

Write [di] (7 times) →

Write [de] →

← datagram reply

3.2.1.4 Read byte

75 Read [d] →

← datagram reply

← Read Reply [d]

datagram reply →

3.2.1.5 Read 64 bytes

80 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

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

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 →

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.

The transfer size for stream access is negotiated. By requiring a transfer size equal to or smaller than the memory write 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, they may still be useful in that case.

Table of Contents

Title..... 1

Section Title..... 2

Section Title..... 2

