OpenLCB Technical Note	
OpenLCB-CAN Frame Transfer	
Oct 15, 2010	Preliminary

1 Introduction

This Technical Note contains informative discussion and background for the corresponding "OpenLCB CAN Frame Transfer Standard". This Technical Note is not normative in any way.

The Frame Transfer layer of the OpenLCB-CAN stack lives above the CAN Physical Layer, and below the Message Network layer. It is responsible for ensuring the reliable transport of the CAN frames that make up OpenLCB-CAN messages. CAN provides reliable transport between any two nodes on a CAN segment within limitations:

1. Every CAN header must be unique by construction. Collisions between frames with identical headers and no data can result in lost frames; collisions between frames with identical headers that do contain data do not result in well-defined behavior. Both cases must be avoided by design to avoid intermittent problems.
2. CAN does not provide any indicator of which node sent a particular frame. Any node can send any frame, and the receiving nodes have no CAN-provided mechanism for identifying the source.
3. It's useful to have a way of addressing a frame to a be processed by a specific node, but all CAN nodes receive every frame.
4. CAN does not provide a "link up" or "link down" notification. Nodes may come and go from a CAN segment at any time, and on an individual basis. In general, one can't assert that a particular node is always present, always comes up first, or never comes up first.
5. On a busy segment, CAN frames are sent in a strict priority order, with the priority determined by the content of the frame headers. If multiple nodes have frames to send at the same time, the highest priority (lowest numerical value header) frame will be sent first. There is no concept of "reserved bandwidth" except through frame priority, and low-priority frames may require significant time to be sent.
6. Nodes must be able to accept frames at the full CAN rate. Hardware filters must be usable to help with that, but even then nodes may receive adjacent frames addressed to them.

Putting a unique source ID field in each frame ensures that the frames are unique (item 1 above), and provides a frame-level indicator of which node sent the frame for the purposes of monitoring, debugging, etc (item 2). Providing a destination address field in the frame header allows a standard method of addressed communications (item 4). Using a distributed algorithm to determine the value for that ID field prevents needing a single "manager" to provide it, thereby handling item 4. The header bit fields discussed below are organized to handle items 5 and 6.

OpenLCB-CAN is required to work with standard CAN components, e.g. bridges and computer adapters must not require customization to operate with OpenLCB. This provides a stringent requirement on protocol design, in that OpenLCB-CAN cannot require specific timing, deliberate creation of error cases or specific error handling.

1.1 CAN background

Standard header frames are also known as CAN 2.0A frames. Extended header frames are also known as CAN 2.0B frames.

A CAN node can emit an overload frame when "Due to internal conditions, the node is not yet able to begin reception of the next message. A node may generate a maximum of two sequential overload frames to delay the start of the next message." (That's 17 to 23 extra bit times each) (from http://rs232-rs485.blogspot.com/2009/11/can-bus-message-frames-overload.html) Image: http://3.bp.blogspot.com/_ycHwJEosotY/SvoMDJMGF7I/AAAAAAAAA64/3Ql4_UQnoBg/s1600/Overload%2BFrame.JPG

People will occasionally refer to a restriction against having seven 1 bits in the top 11 bits of the CAN header. There is no such restriction, and the statements are the result of a misunderstanding. There is a restriction against having seven consecutive recessive bits on the CAN segment, but sending seven consecutive 1 bits will not result in this. CAN uses a bit-stuffing technique to prevent that by inserting a dominant bit on the line after the fifth consecutive recessive bit, and removing it at the receiver.

1.2 Load-Related Frame Synchronization

CAN networks exhibit a behavior called "load-related frame synchronization" (it also goes by other names).

Because of the CAN priority arbitration structure, the network can run at high utilization rates. Under those circumstances, all nodes attempt to send their highest-priority frame, and the node with the absolute highest priority frame (first dominant bit) wins. All other nodes wait until just after that frame and try again.

This behavior greatly increases the probability that nodes with frames ready to send containing the same header will see a collision.

Consider 5 nodes each of which want to send the high-priority frames 1,...,5, plus one more node A that wants to send the low priority frame 15. If all those nodes attempt to send at once, the frames will arbitrate out and will be sent in the order 1,2,3,4,5,15. If another node, B, also wants to send a frame 15, and starts to send it at any point during that entire sequence, it will end up colliding with the 15 that is being sent by A.

Particularly during layout startup, when lots of nodes have frames to send, this can result in cascades of errors that prevent proper operation unless the protocols have been designed to have as few header collisions as possible, and no data collisions.

2 Annotations to the Standard

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

2.1 Introduction

2.2 Intended Use

The OpenLCB protocol suite uses 6-byte Node ID values, but this layer of communications could be valuable in other contexts too. The Standard is therefore written so that any Node ID length from 12 bits through 7*12 = 84 bits can be used in a straight-forward manner.

2.3 References and Context

Note that we reference the Node ID standard, but only to the extent that each node is required to have a unique Node ID. The Standard would work equally well with a Node ID that was allocated via any mechanism, and with any length from 2 to 8 bytes.

2.4 Frame Format

OpenLCB uses a straight-forward transfer of frames, no special features, to reduce complexity and increase robustness.

The Standard is silent on the uses for standard CAN (11-bit header) frames. The proper-operation requirement is so nodes will tolerate them in case they're needed for other purposes. The Atmel bootloader[i] uses standard frames, for example, and the Microchip bootloader[ii] can use them.

This proper-operation requirement could also have been phrased as “shall ignore ...”, but the current phrasing is thought to be more exact.

We don't use RTR frames in the protocol because CAN semantics require a specific use for it, which has been built into some silicon implementations. There are also some arbitration issues, see: CiA Application Note 802 (2005) and http://www.thecanmancan.com/?tag=rtr for more information.

The protocol does not specify transmission of overload frames because not all CAN controller hardware can deliberately send them. We require that nodes be able to handle them because some CAN controller hardware will occasionally send them automatically.

The CAN format has been allocated on nibble boundaries to make it easier to e.g. read dumps of packets. The header is considered to be right (LSB) aligned.

The reserved first bit (MSB) is likely to be used as a priority-boost bit in the future, e.g. so that a gateway can gain priority access to the CAN segment for various operations that require synchronization or atomicity. CAN encodes “do first” priority as 0 and “do later” as 1, so the Standard requires that a 1 bit be sent. To preserve the utility of this, nodes must not require either a 1 nor 0 on receipt.

The order of these fields is chosen to get proper priority and access disambiguation via CAN's standard mechanisms.

Putting the destination alias in the header allows filtering on it with common CAN hardware.

2.5 States

In the Inhibited state, a node can only communicate on the CAN segment to allocate its Node ID alias. This involves sending CIM and RIM messages with a tentative Node ID alias value. Once that process is complete, the node has an assigned Node ID alias and can transfer to the Permitted state, where all communications are possible.

If a node fails, it should restart in the Inhibit state.

2.6 CAN-specific Control Messages and Interactions

2.6.1 Control Message Format

RIM, CIM, etc are assigned low numbers to give them higher transmission priority.

The coding is done to allow up to 7 separate CIM sequence numbers, for use by protocols with longer node IDs.

2.6.2 Interactions

Communications protocols are about more than messages. The protocols must also specify the interactions in which the messages are used.

2.6.2.1 Reserving a Node ID Alias

For example, four CIM frames and a RIM:

0x17123FED

0x16456FED

0x15789FED

0x14ABCFED

0x10700FED

The red nibbles show that these are four CIM and one RIM frames. The blue nibbles are the source ID alias being tested and allocated. The yellow nibbles are the underlying Node ID of the node, in this case 12 34 56 78 9A BC.

The message sequence number MMM is send in descending order, 0x7 to 0x4, to give priority on the CAN segment to messages later in the process. This lets some nodes complete as early as possible.

The CIM/RIM algorithm for finding aliases is designed to work without causing any CAN errors. Arbitration conflicts are OK, but deliberate errors are not, because different CAN implementations may deal with them differently. During execution of the algorithm, some frames might have the same header. They therefore have to have the same contents, or else a non-arbitration error can occur.

Although it might be a useful debugging tool, putting the full NID in the data portion of the frames can cause errors. Therefore, we decided not to do that.

Having NIDa, Nn and TestN in the message ensure that CAN arbitration will work without error.

Keep listening until your CAN hardware says your last CIM message has been sent and the CAN link is idle, plus 500 msec. If you cannot detect that condition, wait 1 second.

- If the RIM (??) message fails with an error occurring during the data portion of the message, it's possible that two nodes have the same NodeID, a fatal error. To prevent the OpenLCB being taken down, this node should stop trying to allocate an alias and signal the error to the user via some non-CAN method, e.g. LEDs, emitting smoke, etc.
- If any other error occurs, repeat from A.

While in normal operation, if you receive a CIM with your source NIDa, send a RIM, which will cause the other node to back off and try something else.

It should not be possible to receive an RIM with your allocated source NIDa in normal operation. If you do, log an error (on whatever it is they log it on there) and restart the process at (A).

If a node sends out an check message containing just the alias, then it could expect that another node to complain if it has the same alias. This would work, except in the (admittedly unlikely) case where both nodes send out the identical check message simultaneously. Neither would recognize a conflict and both would consider that they own the same alias. Therefore, a method needs to be found that guarantees that the check message(s) are unique, even if they are sent simultaneously.

(timing of serialization of low-priority CAN nodes at network startup)

Unfortunately, limitations of CAN will not allow the full 48-bit NodeID to be included in the check message. Doing so would violate CAN rules as it could not be wholly contained in the header, and packets with identical headers are not allowed to have differing data-parts.

Therefore, the CIM/RIM method splits the NodeID into byte-sized parts, sends out 4 check messages (CIM = Check Id Messages, RIM = Reserved Id Message), each consisting of the alias and one quarter (12 bits) of the 48 bit id. Only if all of the CIMs *do not* conflict with another node (ie no RIM is received) does it guarantee that the alias is not in use by another node. Even if two nodes send out the identical packets simultaneously, at least one of these will differ due to a differing ID-part between the two nodes.

CAN arbitration reliably avoids collisions between frames with unique headers. It does not guarantee arbitration between frames with identical headers and no data; if the timing is right, they may overlay each other such that only one frame appears to have been sent. It is very difficult to ensure that two nodes send initialization packets only at different times. In addition, CAN will eventually signal an error if two packets with the same header and different data payloads collide, but not all CAN interface hardware provides reliable indications about why that error occurred. *(lock step issue at startup)*

At the highest level, this algorithm is broadcasting the complete Node ID and tentative alias to see if any other node is checking the same tentative alias. If two nodes have taken the same tentative alias, at least one of the four packets used in that broadcast will not be identical between the two nodes, because their six-byte Node IDs are different in at least one bit. CAN will successfully arbitrate this, and the other node will receive the frame, causing it to back off.

CAN transmissions are not atomic operations; you can receive a frame between the time you tell the hardware to send a frame and the time that frame is actually sent. It's therefore possible for both nodes contending for the same alias value to back off and try again. Using the pseudo-random number generator as a sequence number makes it likely that the next attempt will be made using different alias values in the two nodes. The use of the sequence generator also makes the arbitration process faster in the case of sequential Node ID values. People like to use small and sequential numbers for things, and the sequence generator maps those to very different values that are less likely to collide during arbitration.

The delays at the end of the algorithm are to ensure that higher latency nodes, such as software nodes working through USB convertors, can reply to CID messages from nodes that come up on a working link. OpenLCB is a soft-real-time system, and software that's interacting with it needs to have response times of a couple hundred milliseconds or better to be reliable. The algorithm provides a wait of a few times that to enable those programs to take part.

2.6.2.2 Transition to Permitted State

Note that there is no requirement that this alias be consistent from one run to the next.

2.6.2.3 Node ID Alias validation

This is intended for e.g. late arriving nodes, particularly gateways or diagnostic nodes, so they can get an entire map of a single segment.

Regular nodes won't need to use this. It provides a way for a node joining a segment to rapidly learn the Node IDs and Node ID aliases of the reachable nodes. This may be useful for e.g. a gateway that is connecting two segments that are already in operation, or a monitor node who wants to determine that all nodes have come up properly.

In order to acquire the map between CANids and NIDs, a gateway needs to be able to send a CAN frame requiring "everybody reply with your NID". Much like Ethernet gateway mapping, this needs to return the NID of just the specific CAN-attached hardware, not all the NIDs that can be e.g. reached through a gateway, so it needs to be a segment-specific message defined only at the CAN level. It must not be a OpenLCB common message.

2.6.2.4 Transition to Inhibited State

This is a way to release an alias, while informing everybody that the node is no longer reachable.

The alias is only released if this node allows some other node to allocate it via CIM/RIM messages. The current node can still hang on to it if desired.

Regular nodes won't need to do this. They stay active until this are go instantly offline due to e.g. power-off.

Alias Map Reset - indicates that the local alias in the source address field may no longer be associated with the same NID, and mapping tables should be reset.

Simple nodes don't need to keep track of the Node ID to alias mapping, because they simply respond to frames they receive that contain an alias. Simple nodes just work with the alias itself. They also don't need to track other nodes state, etc.

Gateways will also have to obtain unique aliases for remote nodes they are proxying on to the segment.

Gateways maintain the mapping between remote node's NID and local alias. If they need to break that mapping, e.g. because they need to reuse the local alias due to resource limitations in the mapping tables, they must send a "Mapping Reset" message to force nodes to drop their alias information. *"Mapping Reset" is a CAN-specific message limited to the segment, but all gateways on the segment must act on it.*

Gateways may not be able ensure permanent validity for alias to remote NIDs. For example, if they have a limited routing table, they may need to reuse local alias. Before reusing them, they have to send a "Mapping Reset" frame to drop references to the old NID, followed by a "Initialization Complete" when the alias is allocated to a new NID.

2.6.2.5 Node ID Alias Collision Handling

During normal operation in the Permitted state on a single CAN segment, a node should never encounter a Node ID alias collision.

They can happen when network topology changes, for example when connecting two operating CAN segments. In that case, nodes on the two segments have assigned aliases independently, and can have used the same value. The underlying Node IDs do not collide, because they are assigned globally uniquely, but that's not sufficient to ensure that the shorter aliases are independent.

In the case of a collision, one or both nodes will release their current alias and drop back to Inhibited state. They then can, but are not required to, return to Permitted state with a new alias. In the process they will emit a “Alias Map Reset” and “Alias Map Definition” message that will allow any gateway(s) to update their maps.

Simple leaf nodes don't keep internal mappings between full node IDs and aliases, so this transition will not cause any problems for them unless they are currently in communication with another node, e.g. in the middle of a sequence of frames for a datagram or stream. In that case, the the node may have to invoke higher-level error handling to cause the sequence to be retried.

2.6.3 Node ID Alias Generation

We have a preferred implementation for node ID alias generation which is described in Section 6 “Preferred Alias Allocation Method” below.

The Standard is written to specify the minimum needed to ensure proper operation. This allows developers to create other implementations if needed.

Two other criteria that were considered:

- The sequence of alias values generated by a node shall depend on the node's node ID in such a way that the sequence differs from the sequence generated by every other node.

- Given sufficient iterations, the sequence of alias values generated by a node shall include every valid alias value.

2.7 OpenLCB Message Frame Format

Format selected to allow filtering on destination address, see later section on filter use.

3 Gateways and segment management

(Need to discuss the case of two segments being joined, to discover they've both arbitrated the same aliases, including with gateways on one or both)

Discuss the use of repeaters, bridges and gateways w.r.t. Timing; ref physical layer. Possibility of reordering. Refer to glossary?

If/when a full NID is needed, it can be obtained by sending a "Verify Node ID Number (Addressed)" message with an appropriate Destination Node ID in local alias form. The reply will eventually come back with the Source ID in local alias form and carrying the full NID in the message content.

4 Throughput and Temporarily Deaf Nodes

OpenLCB requires that CAN-attached nodes be able to handle the full frame rate on the CAN bus. There is no guarantee that frames for a given node will arrive with non-zero time between then.

Some nodes are not always able to process frames. For example, the node may cease processing for a short time while writing non-volatile memory. Higher-level protocols, e.g. configuration write datagrams, have mechanisms built in to prevent overrun while writing configuration memory, but there could be multiple activities going on in parallel that will result in frames arriving while the node is not processing.

Short outages can be covered by CAN hardware buffering, so long as the node will eventually catch up even at full arrival rate. Outages long enough that frames are lost due to e.g. buffer overflow require node to broadcast that it's back up if hardware detected frame lost. This is because tehre could have been frames of some form that modified the OpenLCB node state, which are now lost (e.g. drop alias, CIM/RIM in absence, or even higher level events)

29-bit extended header and 8-byte payload results in a total transmission taking about 135 bit times, varying slightly with bit-stuffing.

125Kbps with max-length extended header frames is about 900 frames/second. CAN is very good at running at 100% utilization, so long as nodes can keep up and a proper set of priorities is in place.

CAN is designed to run at 100% usage without problem, and it's routine for CAN segments in other contexts to run at 100% for extended times. Therefore, the CAN network does not require any inter-frame spacing, limitations on bandwidth usage, etc; the segment can be allowed to run at 100%.

The real limitation is whether the nodes themselves can handle the full rate of frame arrival. There is (currently) no mechanism in OpenLCB to throttle the rate of frames arriving at a node, nor is it easy to create one in a CAN network. The stream and datagram protocol have been designed to allow a node to efficiently use a very limited amount of buffer memory, but don't provide mechanisms to limit the arrival of frames.

For long term reliability, a node must be able to completely process an entire CAN frame in the time it takes to receive the next one, which may be as little as 42 (?) bit times.

Eventually OpenLCB may have to discusse use of Overload Frames to throttle for e.g. NVRAM writing, but the hardware may also be doing this.

Nodes must handle full rate messages, specifically including messages not addressed to them. (The datagram and stream protocols provide ways for nodes to indicate whether or not they have sufficient buffering for the next transmission, triggering retries as needed) For long term reliability, a node must be able to completely process an entire CAN frame in the time it takes to receive the next one, which may be as little as 64 bit times, or about 500 microseconds.

5 On the choice of a 12-bit alias length

(Should this be here? Or elsewhere? In either case, clean it up)

How big should the NodeID alias field be on CAN links?

- Smaller (fewer bits) allows more payload and/or simpler coding of other parts of messages.
- Larger allows more unique nodes to be accessed.

5.1 Issues

5.1.1 Addressing

The Node ID size limits the number of unique nodes that can take part in communications on the CAN segment. Because Node ID aliases are assigned independently on each CAN segment, the only issue is how many different nodes are involved, not which ones they are or what pattern(s) are available in their address numbers.

For electrical/timing reasons, a segment itself can only support a maximum of about 50 nodes, but communication also involves nodes off the segment. For example, chaining N CAN segments via bridges will generally make up to 50*N nodes available. We have a use case of chaining small numbers of CAN segments together, implying the need to address 500 or 1000 nodes.

As another example, a very large network might connect many CAN segments via TCP/IP or other networked connections. Each CAN-attached node might need to communicate with a larger number of others spread across that network.

(CAN backbone case)

In some cases, want 2 aliases (source and destination), plus a few more bits, in 29 bits total. Using 12-bits for each address leaves 29-(2*12) = 5 bits for these purposes, which seems about right.

5.1.2 Collisions during Allocation

(not talking CAN arbitration here, but two nodes trying to use same alias) The allocation process resolves collisions (attempt to use an already allocated Node ID alias), but that takes time. Minimizing the number of collisions is good. If the address space is just the size of the number of things you want to address, the collisions get hard /slow to resolve across multiple nodes. This points to having a factor of 2, at least, between the number of nodes to be addressed and the size of the address space.

(Mostly talking about separate nodes here, not just one big interface node: Points to CAN backbone case)

5.1.3 Size

Streaming provides the most stringent test of alias sizes. We'd like to have all 8 bytes of CAN frame payload available for transporting stream data, which requires putting both source and destination addresses, plus whatever control information is needed, into the header.

To get the full payload (8 bytes) for streaming requires getting the rest of the protocol control, including two aliases for source and destination, in the 29 bit header. Allowing 1 bit for priority/extension, 1 for protocol control, 3 for MTI and stream control (all very bare minima) leads to a 12-bit node ID alias length.

6 Preferred Alias Allocation Method

This section documents the preferred algorithm for generating unique Node ID aliases.

Requirements

- Must not require any user intervention; no user presets required or permitted
- Must be possible to map the assigned alias to a Node Sequence Number and vice-versa.
- It is not required to come up with the same alias every time a node or segment is powered up.
- Must always converge to an alias, regardless of when or in what order nodes come up.
- It is desirable that a node watching the process must be able to monitor how the decision was reached and what the result was.
- It is desirable to detect duplicate (non-unique) NIDs, but 100% reliable detection is not required.

6.1 Recommended alias generation algorithm

The 12 bit alias has to be derived from a 48-bit sequence number in a way that keeps as much entropy as possible.

PRNG does this; runs through full 48-bit sequence; each node joins at separate point in the sequence, so the 48 bit values are unique. Want to have 12 bit version, so need an algorithm that preserves that to the extent possible.

Sequentially numbered nodes will have initial seeds that differ only in the low bits of the PRNG output. For shift-register PRNGs, this can present a problem, as all bytes of the result are shifting together. (Traditionally, shift-register PRNGs shift toward the LSB) Only one bit changes. Combining bytes (or even 12-bit chunks) via XOR or add operations results in reduced entropy.

The OpenLCB proposed PRNG (next section) uses add operations with a constant term to avoid this problem, so treating the 48-bit PRNG output as four 12-but values, which are then XORed, works well.

6.1.1 Preferred PRNG

The proposed 48-bit PRNG is from "A 48-Bit Pseudo-Random Number Generator", Heidi G. Kuehn, Communications of the ACM, Volume 4, Issue 8 (August 1961), pages 350-352.

$x_{i+1} = (2^9+1)\ x_i + c$

where c = 29,741,096,258,473 or 0x1B0CA37A4BA9.The paper actually describes a PRNG that uses signed arithmetic, but for our purposes the 48th "sign" bit isn't important. *(verify that about sign) (This is a byte+bit shift and add, so it's quick to calculate)*

Note that, unlike some other PNRGs, this one can generate a zero result, and can accept a zero seed.

It is initialized with the node ID as a 48-bit key, with the most significant byte (first byte) of the node ID as the most-significant byte of the PRNG value.

6.1.2 Preferred PRNG to node ID alias mapping

The 12-bit node ID alias is made from the 48-bit PRNG value by splitting the PRGN value into four 12-bit values and XORing them together. This preserves the information content of every bit in the final result.

6.1.3 C implementation[iii]

The PRNG state is stored in two 32-bit quantities:

```
uint32_t lfsr1, lfsr2; // sequence value: lfsr1 is upper 24 bits, lfsr2 lower
```

Load the PRNG from the Node ID:

```
lfsr1 = (nid[0] << 16) | (nid[1] << 8) | nid[2];
lfsr2 = (nid[3] << 16) | (nid[4] << 8) | nid[5];
```

Step the PRNG:

```
 // First, form 2^9*val
 uint32_t temp1 = ((lfsr1<<9) | ((lfsr2>>15)&0x1FF)) & 0xFFFFFF;
 uint32_t temp2 = (lfsr2<<9) & 0xFFFFFF;

 // add
 lfsr2 = lfsr2 + temp2 + 0x7A4BA9l;
 lfsr1 = lfsr1 + temp1 + 0x1B0CA3l;

 // carry
 lfsr1 = (lfsr1 & 0xFFFFFF) | ((lfsr2&0xFF000000) >> 24);
 lfsr2 = lfsr2 & 0xFFFFFF;
```

Form a 12-bit alias from the PRNG state:

```
uint16_t LinkControl::getAlias() {
  return (lfsr1 ^ lfsr2 ^ (lfsr1>>12) ^ (lfsr2>>12) )&0xFFF;
}
```

7 CAN Controller Filters

This coding has been generated such that simple nodes can use three hardware filters to select only frames that are of interest to them.

Purpose	Mask	Required Value
CAN-specific control messages	0x0800,0000	0x0000,0000
OpenLCB format addressed to this node	0x0C00,0FFF	0x0C00,0nnn
Other simple OpenLCB format MTIs (See later doc)	0x0F00,0000	0x0800,0000

(but the simple-MTI stuff is at Message level)

8 Additional References

B. Gaujal and N. Navet, “Fault confinement mechanisms on CAN: analysis and improvements”, IEEE Transactions on Vehicular Technology, pp.1103-1113, 54(3), May 2005 (An interesting analysis of how CAN nodes react to error rates on the CAN bus by e.g. taking themselves offline, etc)

(Take some from physical layer doc)

“A 48-Bit Pseudo-Random Number Generator”, Heidi G. Kuehn, Communications of the ACM, Volume 4, Issue 8 (August 1961), pages 350-352.

Sections of Etschberger book

Table of Contents

i See Atmel application note “AVR076: AVR® CAN - 4K Boot Loader” http://www.atmel.com:80/dyn/resources/prod_documents/doc8247.pdf

ii See Microchip application note “AN247 A CAN Bootloader for PIC18F CAN Microcontrollers” http://ww1.microchip.com/downloads/en/AppNotes/00247a.pdf

iii The code in this section is from the OpenLCB C sample implementation available at http://www.openlcb.org/trunk/prototypes/C/libraries/OlcbCommonCAN/ The specific code is available at http://www.openlcb.org/trunk/prototypes/C/libraries/OlcbCommonCAN/LinkControl.cpp