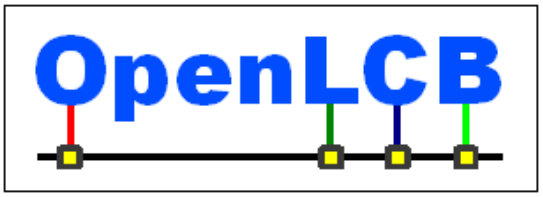

OpenLCB Technical Note	
Firmware Upgrade	
~~Feb 6, 2016~~Feb 6, 2016	~~Proposal~~Draft

1 Introduction

OpenLCB hardware contains a significant amount of complexity, both in terms of the protocol stack involved, as well as the node-~~-~~carrie~~ds certain~~ descriptors which can be considered a user interface. The necessary code and data is stored in non-volatile memory of the node, and it is highly desirable to have this non-volatile memory field-~~update~~able.updatable. There may be many reasons for a new released version of firmware ~~released~~: fixing issues discovered after shipping the node, adding new features to the node, keeping up to date with the evolving standards, or simply to change the language of the user interface descriptors.

There are many desirables for a firmware update mechanism that is expected to be operated by the end-user:

- The update should be safe and reversible. The end-user must not be able to irrecoverably damage (aka *brick*) the node with the update process. Failures during the download process are possible but must be recoverable by re-trying the update process.
- The update should be reasonably fast. With the largest OpenLCB deployments expected to range up to hundreds of nodes, the update process per node should not exceed a few minutes.
- The update process should be standardized so that the ability to update nodes can become a feature of all (or most) commonly available Configuration Tool softwares. This allows the user to perform the firmware upgrade tasks using the existing tools that they are familiar with. A worst-case scenario would be if every manufacturer has to implement and release separate tools for updating their nodes (for all possible host operating systems), and the user having to acquire and learn using all these new tools; and every manufacturer having to go to the effort to implement, deploy and support them across a wide variety of environments.
- The node should be able to check and validate the firmware being downloaded. The exact detail and strength of this validation shall be left at the discretion of the manufacturer and cannot be dictated by the standard. Most manufacturers will want to check at least the compatibility of the new firmware with the particular node hardware, but some may want to have significantly stronger form of security, including using strong cryptography to avoid reverse engineering or modifying the firmware being downloaded.
- The firmware update process should be user-friendly and should happen through the OpenLCB bus; preferably without the need to physically access the node. Since the node may be mounted in a hard-to-access location under the layout, but the network browsing tools have sufficient information to locate and identify the node using standard OpenLCB

protocols, the end-user has full ability to target and select the specific node that they want to update.

The most important requirement is that of *safety*, i.e., the user should not be able to brick the node while attempting an update. This is very difficult to achieve, because the update process will inherently modify the node's program. A protocol error (aka implementation bug or incompatibility), or everyday events such as loss of power or network connectivity during the update process will likely leave the Target Node with a corrupted firmware, a situation from which it is notoriously hard to recover.

The only reasonable way to recover from a corrupted firmware is to employ a *double-firmware* approach, where the node is shipped with two firmwares, one that is ~~updateable~~updatable for regular operation (Primary firmware), and another whose sole purpose is to update the Main firmware (aka Bootloader firmware). Of the two firmwares the node would typically start up with control being held by the Bootloader firmware, which checks that the Main firmware is correct, matches the hardware, and not corrupted before yielding control to it. If any of these checks fail, the Bootloader firmware takes control and starts up the node, participating in the bus activity only as far as necessary to receive the correct firmware data file.

The obvious drawback of the double firmware approach is the size and cost associated with the secondary firmware. A minimalistic Bootloader firmware can however be implemented in 5-8 kbytes of program space, which will not add a prohibitive cost to the node hardware on top of the regular firmware, which is expected to be ~~64~~32-128 kbytes in size.

Despite the above recommendation, the actual firmware upgrade protocol described in the standard is agnostic to the actual implementation, therefore it is the manufacturer's choice whether they implement the firmware upgrade process with double firmware, or some other method.

1.1 Served Use Cases

The user checks the manufacturer's web-site and finds that an updated firmware is available. The user downloads a binary file from the manufacturer containing the new version. Then the user launches the Configuration Tool (CT), selects the particular node in the network browser that she wants to update, and selects the Firmware Update option in the user interface of the CT. They~~She~~ point~~s~~ to the recently downloaded file. The CT starts the download process and shows a progress bar as the transfer is happening. After the download is complete, the CT automatically reboots the node, which starts up with the new firmware. The user can now go to the configuration panel of the node and start using the new features that are in the new version of the firmware.

If the manufacturer decides, a more powerful node of the given manufacturer (such as a Command Station) may contain firmware versions of other nodes of that manufacturer. If desired, the powerful node could enumerate all nodes on the network, and automatically update the ones that have an outdated firmware. It is up to the manufacturer's discretion on whether or not to implement such a feature and how to manage the customer expectations of seamless operation, for example by automatically migrating the configuration of the upgraded node, or asking for user confirmation before the upgrade process.

1.2 Unserved Use Cases

This protocol does not solve the problem of discovering what hardware node requires which firmware, and does not help the user to locate and acquire the new firmware data file for any given node. There is

no check for validity or correctness of the firmware data file in the protocol or the CT ~~(~~ ~~that~~it is left to the node developer~~)~~ to perform these checks.

There is no inherent protocol support for bulk updating many nodes at once. Even though the data transfer could be observed by many nodes participating on the bus (especially if a shared physical media bus, such as CAN is used), the complexity involved in multicast transfers heavily outweigh the benefits of updating many nodes at once. The firmware update process is define~~d~~d ~~d~~so that it is possible to perform without user interaction, directly through the bus, so it is easily scriptable. If ~~there will be~~ a high demand arises for bulk updating many nodes, then ~~the~~ common CT software packages ~~will~~may ~~offer~~implement a feature for updating many nodes sequentially (or in parallel) without the user being present at the terminal.

The Firmware Upgrade standard also does not define what happens with the configuration data during and after the update. The CT does not have a responsibility to translate or update the configuration memory contents in case of an incompatibility between the old and the new firmware. The firmware (either the Primary or the Bootloader) is expected to perform reinitialization or migration of the configuration memory contents as required. The reason for this decision is that such a translation would be very difficult to specify and could easily lead to feature bloat or a Turing-complete language interpreter being required to be present in all CT software packages, which we deem an unnecessary complexity.

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 References and Context

There is no additional commentary to this section.

2.4 Message Formats

There is no additional commentary to this section.

2.5 States

There is no additional commentary to this section.

2.6 Interactions

2.6.1 Definitions

The Configuration Tool is typically a computer software that is connected to the OpenLCB bus. The user can download the new firmware data file to this computer, and use the user interface of the Configuration Tool to browse the nodes available on the bus, select the node to upgrade and initiate the firmware update process.

There are interesting alternative approaches to the role of the Configuration Tool.

- For example a smart throttle may receive the firmware data file using a USB memory stick plugged into a USB port, and use the user interface of the throttle to select the target node to update and perform the firmware update process. This way customers who do not have a computer connected to their OpenLCB bus can still perform firmware upgrade, provided they have an appropriate smart throttle. A smartphone application when acting as a smart throttle could even acquire the new firmware data file from the Internet.

- Another example would be an internet-connected service operated by the hardware manufacturer. The customer would connect the OpenLCB bus using a TCP/IP-based OpenLCB segment to the manufacturer's server. The manufacturer's server would then enumerate the nodes on the OpenLCB bus, select those that are in need of a firmware update, and automatically perform the firmware update process with the latest firmware.

- Such an automatic firmware update mechanism may also be implemented in a smart throttle (or any other node with sufficient storage) provided by ~~a~~the given manufacturer, if the manufacturer deploys it with the necessary node firmware files.

2.6.2 State transitions

The two states of the node (~~Bootloader~~Firmware Upgrade and Operational) operate with a dramatically different feature set, and most information that may be acquired or cached from one state (such as a Protocol Support Reply) will not apply to the other state. Therefore the node switching between the two states must signal the other participants of the bus that all such cached information shall be discarded. This can be done by appearing to reboot the node, which is visible on the bus by a Node Initialization Complete message. The requirement of entering Uninitialized state to transition between ~~Bootloader~~Firmware Upgrade and Operational states forces the node to emit a Node Initialization Complete message.

A node that is in ~~Bootloader~~Firmware Upgrade state will appear as in-operational to the customer. Therefore it is important that a node does not accidentally enter this state. In all normal conditions the node shall be in Operating state. Normal conditions include power-cycling the node, and thus the node must reach Operating state upon powering up.

The exceptions are granted for two purposes:

- If the node has its production firmware missing or corrupted, then reaching an Operating state is likely impossible. The node may detect this upon power up and start in ~~Bootloader~~Firmware Upgrade state. This allows the user to use a Configuration Tool to download an appropriate firmware and restore the node to full working operation. An important case when firmware corruption may occur is if the firmware download process was interrupted due to power loss or hardware error. Another case when this may happen is if the user downloads an incompatible firmware whose incompatibility was not detected during the download operation. A node may also detect cases of irreparable software errors (for example the production firmware crashing every time upon startup) and power up in ~~Bootloader~~Firmware Upgrade ~~mode~~state, expecting the user to contact the manufacturer and supply a working firmware using the firmware update method.

- The customer should be able to physically force booting the hardware into ~~Bootloader~~Firmware Upgrade state. In case the production firmware is misbehaving to the extent that entering ~~Bootloader~~Firmware Upgrade state using a Configuration Tool is impossible (e.g. due to a bug in the operating software), this option gives the user a physical recovery mechanism. While not really convenient, as physical access to the node is required, it still allows the user to recover "bricked" hardware nodes, so long as the firmware needed for ~~Bootloader~~Firmware Upgrade state is intact. It is not required by this standard for the node to come with a hardware switch, if the manufacturer decides that this option is not necessary for their hardware.

The remaining state transitions describe how a Configuration Tool may request a node to change to a specific state. Note that the user may also perform these transitions without a CT: to transition from ~~Bootloader~~Firmware Upgrade state to Operating state it is enough to power cycle the node, and to transition into ~~Bootloader~~Firmware Upgrade state, the user may use the hardware switch while power cycling the node.

~~It is important that t~~The CT ~~has no real means to~~may detect whether a given node is in ~~Bootloader~~Firmware Upgrade state or in Operating state by examining the Protocol Support Reply. ~~Therefore~~Nevertheless, for the simplicity of the CT implementation~~,~~ the state transition is defined in a way that it is legal to request transition to ~~Bootloader~~Firmware Upgrade state even if the node is already in ~~Bootloader~~Firmware Upgrade state, and the CT will receive the same reply to it.

2.6.3 Data Transfer

The data transfer is defined as sending a file as-is from the manufacturer to the node. The file must be sent binary-accurately. This allows maximum freedom to the manufacturer in defining the format and contents of the file, while also keeps the standard and the CT implementation complexity at a minimum. More discussion about the file format is at Section 3.1. The transfer is also required to start at offset zero, because the offsets are not required to match any physical hardware address in the Target Node – most manufacturers would want to prefix the binary data with at least a header to check if the firmware is intended for the given hardware it is being downloaded to. Making offset translation is not a significant complexity in the bootloader implementation.

Such a header may cause the Target Node to reject the payload being transmitted, and both ~~the~~ the Stream and the Datagram / Memory Configuration protocol allows this. The target node has to return a Stream Data Abort message to the CT, supplying an error code for the failure~~.~~, or in case of the Datagram-based transfer a Datagram Rejected message or Memory Configuration Write Reply datagram with the error code. The Standard lists a few error codes in Section 7 ~~[~~(Allocations)~~]~~ that may be used to denote common problems. Of course all other standardized error codes may be used, including the general Permanent Error (0x1000) or Temporary Error (0x2000), albeit these may not be very helpful for the end-user.

If the transfer was aborted, it is possible that the Target Node has an incomplete firmware. In such a case the node may be in~~-~~operational, but it shall not be destroyed permanently, even in case of a power loss. Upon power-up the bootloader firmware shall detect that the main firmware is corrupted, and start up in ~~Bootloader~~the Firmware Upgrade state.

3 Background

3.1 Payload content format

The Firmware Upgrade Standard purposefully does not specify anything about the format and contents of the firmware data file. It is fully up to the manufacturer how the data contents are defined. The only requirement from the protocol is that the bytes be transferred unmodified to the node in ~~Bootloader~~Firmware Upgrade state.

Simple implementations may just supply a stream of bytes to be written to the non-volatile memory of the node. Other implementations may add a header describing the hardware name and version with which the firmware file is compatible, allowing for the Bootloader firmware to cancel the upgrade process in case it detects an incompatibility. More elaborate implementations may create internal headers and packet format inside the binary stream with an arbitrary encoding of the choice of the manufacturer to convey a diverse set of information, such as checksums, offsets at which the data is to be written, or signatures[1]. It is also possible to encode a configuration memory update script inside the firmware upgrade file, if the manufacturer so chooses, and supplies the respective Bootloader with the necessary interpreter.

We describe two interesting options, one for its simplicity and another for its security properties.

- For the simple option the data bytes as they arrive are written directly to flash. The program space is assumed to be one consecutive region, and the compiled and linked binary of the new firmware is written to flash at this offset, which is hard-coded in the Bootloader firmware. There are two notable positions in the firmware file, which contain special data instead of instructions. The first notable position, present within the first writable block of the firmware file, contains a magic code that identifies the hardware type and version. If there is a mismatch, the firmware update operation is aborted before the existing firmware is corrupted by the update process. The second notable position contains the size and checksum of the entire firmware.

  When the node is powered on, the Bootloader firmware starts up, checksums the Main firmware and compares with the value given at the second notable position. Should there be a checksum mismatch, the Bootloader firmware initializes the OpenLCB bus, emits the well-known event with ID “Firmware corrupted”, and awaits a CT to deliver the correct firmware for the node. If the checksum matches, the Bootloader firmware jumps to the entry point of the Main firmware, which will initialize the OpenLCB bus, and the node operations commence as expected.

- In the secure option the data bytes are encompassed in a lightweight framing format. Each frame is of the size that can be held in RAM by the node in ~~Bootloader~~Firmware Upgrade state. The frames are suffixed with a cryptographically secure hash that the node in ~~Bootloader~~Firmware Upgrade state verifies before writing the specific block of data to non-volatile storage. It is also possible to use encryption on the frames, with the Bootloader firmware containing the necessary decryption keys.

  Using this method the manufacturer can ensure that the firmware that gets loaded into their node is authentic and is originating from themselves. This might be important if for example

[1]For example it is valid to have a HEX file be the firmware upgrade data file; the interpretation of the HEX file contents would be up to the receiving node (i.e. the Bootloader firmware). This is not particularly efficient use of bus bandwidth (since we are transferring 4 bits per byte), but certainly a valid implementation of the standard.

different boards with different pricing are based on the same hardware but a different firmware to achieve a richer feature set.

3.2 Alternatives considered

3.2.1 Using Datagram-based Memory Configuration writes

A valid standards-compliant alternative is to perform the data transfer using datagram-based writes of the Memory Configuration protocol.

Datagrams are meant for short data transfers. Based on the global guidelines, if the data to be transferred is more than about 1 KB, streams should be used. Streams perform better buffer management, and require significantly fewer round-trips than datagrams. For each 64 bytes written by the Memory Configuration protocol Write command, the following messages need to be transferred over CAN:

- > Write request, 8 bytes header
- > Write request, 64 bytes data
- < Datagram received OK, reply pending
- < Write response OK
- > Datagram received OK

This sequence requires not only significantly more bytes on the bus, but also two changes of data transfer direction, which involves significant latency by needing to traverse all gateways' and routers' message queues. This latency will dramatically reduce the maximum achievable transfer rate, since it is incurred for every single 64-byte block of data.

On the other hand, streams will allow typically several kilobytes of data transferred before a change of direction for an acknowledgement is required. As an example, a practical implementation traversing a CAN-USB and a TCP-TCP gateway was able to achieve 1.7 kbyte/sec transfer rate with datagrams, and 8 kbyte/sec transfer rate with streams.

An undoubted benefit to the datagram-based transfer is that the Datagram Transport is already an adopted standard, whereas Stream Transport is still in draft. ~~The amount of additional code required in the bootloader for the stream implementation is not significant, since it can replace the need for support of segmented datagram packets.~~

3.2.2 Using a directly initiated stream

It would be possible to eliminate the Memory Configuration protocol from the bootloading process, and allow the Data Transfer to start directly with the Stream Initiate command from the CT to the Target Node.

The stream content could be identified by a specific stream content UID that is reserved for the Firmware Upgrade protocol.

The benefit of this implementation is that less code would be needed in the bootloader firmware. This is questionable though, since the Reboot command would still have to be implemented in the Target Node even in ~~Bootloader~~Firmware Upgrade state in order to facilitate the state transition back to Operating state. Defining a custom message for reboot request would create unwanted parallelism in the OpenLCB protocol stack.

3.2.3 Using custom protocol instead of Memory Configuration

It would be possible to define custom messages for carrying bootloader data.

The benefit would be a potentially simpler implementation of the protocol, thus the Bootloader code being smaller.

However, this messaging protocol would face the exact same requirements and challenges as the Stream protocol, therefore the most likely best solution would be exactly the same. Also, addressed messages on the CAN bus have a payload efficiency of 6 bytes/frame~~message~~, while streams have 7 bytes/frame~~message~~. A custom datagram-based protocol would achieve higher bus efficiency, but at the expense of fully implementing the datagram segmentation in the bootloader, which costs more code space than stream reception.

Defining a custom protocol would also increase the mental complexity of the Firmware Upgrade standard, the engineering cost of implementation, especially that standard solutions of existing stacks (like that of the Stream protocol of the Memory Configuration) could not be used.

3.2.4 Using non-OpenLCB messages

The physical bus can carry other messages than OpenLCB. These messages, however, cannot be routed through OpenLCB gateways, thus there is no standard method of ensuring that the Configuration Tool, which may be on a different bus segment (including even a different part of the world, in case of some use-cases mentioned earlier) can reach the node that is switched to ~~Bootloader~~Firmware Upgrade mode.

For example some MCUs contain embedded ROM firmware that use CAN-bus based protocols for flashing the firmware. The exact protocol is however not standardized across MCU vendors, therefore not a helpful base to build on. There can also be different encodings that make the operation on the same CAN bus segment as OpenLCB impossible. Sometimes even the bit rate would not be matching.

3.2.5 Using a different bus

There can be many alternate methods of supplying data to a microcontroller in a hardware node, for example through USB, UART, SPI, etc. Any such alternative connections would require on one hand specialized hardware to be acquired by the end-user, on the other hand physical access to the board to be upgraded. Standardizing on a secondary peripheral bus would also not be cost-effective to hardware manufacturers. This solution fares as significantly inferior to anything natively OpenLCB.
Certain manufacturers or manufacturer groups using the same MCU technology may decide to expose the native ICSP pins on headers matching the industry standards for the given MCU (for example JTAG headers, AVR-ICSP or Microchip PicKit headers) as an in-house debugging tool that may be used by expert customers with the necessary tools to recover hardware from a state where the bootloader got corrupted. However, when the recommendations of this document are adhered to, there should be no case where a device in the field can corrupt itself to this recovery need.

3.2.6 Structured payload content

Please read Section 3.1 (Payload content format) before reading this section.

An important alternative to consider is to have the firmware file be supplied in a structured file format, and require the CT to implement the parser of this file format, executing a series of operations on the Target Node as part of the firmware upgrade process.

A straightforward choice for such a structured file format would be some variant of the HEX file format, with the address space (including extra address bits commands) defining the memory space as well as the offset to which to write the particular payload data. This would allow one firmware upgrade operation to write to multiple memory spaces, or write to discontinuous areas of memory.

A helpful use-case of this option would be that the firmware upgrade can wipe the configuration memory space. Writing the firmware code to different, discontinuous regions or non-zero offsets does not seem to be a compelling use-case, since offset translation is trivial to do in the Bootloader firmware. Wiping the configuration storage also does not seem to be a critical feature of firmware upgrade, since all production node firmware must support the “factory reset” command which contains code to do exactly this. Therefore making the firmware detect incorrect configuration memory contents and perform the wipe automatically does not seem to be a major burden on the node developer.

The drawback of this approach is that a single file format needs to be agreed upon, and all CT implementations will require code to interpret this file format. There are numerous different, similarly looking but incompatible HEX file formats for example. There would also be a pressure of adding extra features (such as sending checksums ahead of time, sending hardware detection information ahead of time, etc) which would also increase the complexity of the CT. With an opaque binary payload all of these features are the freedom and responsibility of the hardware manufacturer, the CT remains simple and it is easy to support all possible nodes.

In summary, the benefits of this alternative are questionable while it introduces significant complexity and thus we chose the simpler alternative to standardize. The desired additional features are achievable to the manufacturers if they so choose by using the methods outlined in Section 3.1.

4 Detailed interaction flow

This section describes the exact flow of packets between the Configuration Tool and the Target Node. Each message is printed formatted for CAN-bus in GridConnect format for brevity, albeit the protocol is not CAN-specific.

Legend:

- > packet: data sent from the CT to the Target Node
- < packet: data sent from the Target Node to CT.
- The Target Node will have alias 0x4AA on the bus, full node id of 1A.2A.3A.4A.5A.6A
- The CT node will have alias 0x3CC on the bus.

> :X1A4AA3CCN20A1EF~~B~~;	Memory Config datagram: ~~Enter Bootloader~~Freeze Firmware Space (0xEF)
< :X19A284AAN03CC00;	Datagram received OK. This message may be omitted.
< :X171A24AAN; ...	There may be some alias allocation packets here.
< :X191004AAN1A2A3A4A5A6A;	Node initialization complete.

> :X198283CCN04AA;	PIP request to discover node capabilities
< :X196684AAN03CC700010000000;	PIP reply; indicates streams are supported.
> :X1B4AA3CCN202000000000EFFF; > :X1D4AA3CCNFF55;	Memory Config datagram: Stream write to space EF, offset 0, source stream id 55.
< :X19A284AAN03CC80;	Datagram received OK. Reply pending.
~~TODO: finish writeup here~~ :X1B3CC4AAN203000000000EF5A; < :X1D3CC4AAN55;	Memory Config datagram: Stream Write Reply OK, space EF, offset 0 dest stream ID 5A, source stream ID 55.
> :X19A283CCN04AA00	Datagram Received OK, no reply.
> :X19CC83CCN04AAFFFF000055;	Stream initiate request; maximum buffer 64k-1, stream src id 55
< :X198684AAN03CC01008000555A;	Stream initiate reply OK; buffer 256, src id 55, dst id 5A
> :X1F4AA3CCN5AA5456112B50B99; > :X1F4AA3CCN5AB077952FF3138E; > ... > :X1F4AA3CCN5A97A0A4CC;	Stream data for dst stream 5A (total of 256 bytes)
< :X198884AAN03CC555A0000;	Stream continue
...	May add more stream data here
> :X198A83CCN04AA555A;	Stream close
> :X1A4AA3CCN20A0EF;	~~Memory config reboot~~Memory Config Datagram: Unfreeze Firmware space (0xEF)
< :X19A284AAN03CC;	Datagram Received OK. This message may be omitted.
< :X171A24AAN; ...	There may be some alias allocation packets here.
< :X191004AAN1A2A3A4A5A6A;	Node initialization complete.

340

Table of Contents

