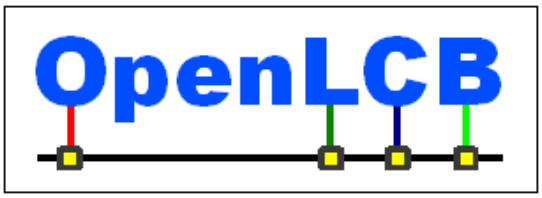

OpenLCB Technical Note	
Configuration Description Information	
Apr 25, 2021	Adopted

1 Introduction

"Configuration description information" in this context refers to *fixed* information available from an OpenLCB device, via OpenLCB, so that other devices can properly and correctly configure it. The information is fixed, so that it can be pre-compressed, stored in the device, and just supplied when needed with minimal work on the part of the device and the device's developers. This means that e.g. the actual current configuration contents are not available as part of the CDI, as that is variable information. Similarly, the CDI cannot contain e.g. a serial number as that would require different CDI contents in each node of a single type.

Other information may be available via e.g. manuals or the Internet, and there may be pointers to that information in the CDI, but the format of that information is not under specification here.

A key use for CDI is to enable a Configuration Tool (CT) to know how to configure the node. The configuration tool will use the CDI information to render some form of suitable Graphical User Interface to allow the user to easily and intuitively configure all aspects of the node's capabilities. An important design choice was to embed the CDI into each node so that the system has all it needs to configure the node without having to source the information externally to the OpenLCB network from the manufacturer or some other on-line repository via the Internet or a CD/DVD etc. While the CT is likely to be a program running on a PC, it could be a hand-held device like mobile phone or PDA or even a custom-built device.

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

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

The Memory Configuration Protocol is one use for the CDI, and one way to retrieve it, but CDI is independent of that. See the OpenLCB Memory Configuration Protocol documentation.

CiA 306 "Electronic data sheet specification" describes the CANopen version of a similar capability.

2.4 Content

"Initialized State" means talking on the OpenLCB, so CDI is fixed while you're up. But it can change when the node e.g. resets.

The CDI information isn't allowed to change while the node is up. A node may not change it after any part of it has been retrieved and before the next transition of the node away from the Initialized state. (The transition back to Initialized state tells other nodes to flush their caches and pick up any changed content.) A node using the CDI can assume that, once it's read the CDI, it doesn't need to read it again until the source node goes offline.

2.5 Format

The format can be extended later for e.g. compression, via changes to the 1st character or some other method.
The choice of XML version 1.0 was made to restrict the XML feature set that can be utilized in the CDI documents. Because the using node may not have external connections, XML features requiring those (Xinclude, schema default values, …) are not be assumed to be present. By specifying XML 1.0 only, XInclude, XML Namespaces, XML Base, xml:id and other extensions have been excluded.
The CDI may specify stylesheets (because the XML 1.0 standard is consistent with that).

The online schema is really the normative thing, because that's what we check for conformance testing. There is no requirement that a node using the CDI to perform validation against the schema. But since the contents of the schema are normative, it has to obey the defaults, etc., for missing elements. There is further information about schema versions in the Section Future Extension.

Hexadecimal notation (0x1234) for numbers is not permitted. Although it may have made the CDI easier for people to read, the XML is primarily intended for consumption by OpenLCB nodes. Providing optional coding, such as hexadecimal numbers, makes those nodes more complex.

2.5.1 XML Elements

2.5.1.1 <identification> Element

There is a basic set of identifiers that allow a user to recognize a Node. These are: the manufacturer name, model name, certain version numbers; as well as a user-definable name and description. The first set of these is called "fixed" node identification information, because it only depends on the hardware (and firmware) of the node. The second set is user-modifiable, or "variable" identification of the node.

There are three ways for a Configuration Tool or a Network Browser tool to access this basic identification information:

- The fixed information may be encoded in the CDI as specific XML tags.
- Both the fixed and the variable information can be available as specific Memory Spaces in a fixed format. This is called ACDI (or Abbreviated Configuration Description Information). This access method allows changing the variable information.
- Both the fixed and the variable information can be fetched with one single message specifically designed for light-weight enumeration of all nodes using the Simple Node Information Protocol.

This Standard specifies the CDI-held information for format and access to the ACDI-held information. The standard also requires that all these different access methods must return the same values.

2.5.1.2 <acdi> Element

ACDI defines two specific memory spaces that contain the basic identification information. The layout of these spaces is fixed in the standard, and allows the Configuration Tool to access these values without needing to fetch and parse the CDI.

This is the equivalent descriptor to the ACDI spaces:

```
<segment origin='0' space='252'>
    <group>
        <name>Manufacturer Information</name>
        <description>Manufacturer-provided fixed node description</description>
        <int size='1'>
            <name>Version</name>
        </int>
        <string size='41'>
            <name>Manufacturer Name</name>
        </string>
        <string size='41'>
            <name>Node Type</name>
        </string>
        <string size='21'>
            <name>Hardware Version</name>
        </string>
        <string size='21'>
            <name>Software Version</name>
        </string>
     </group>
</segment>
<segment origin='0' space='251'>
    <group>
        <name>User Identification</name>
        <description>Lets the user add his own description</description>
        <int size='1'>
            <name>Version</name>
        </int>
        <string size='63'>
            <name>Node Name</name>
        </string>
        <string size='64'>
            <name>Node Description</name>
        </string>
    </group>
</segment>
```

Manufacturers are encouraged to put these segment definitions into their CDI as well, especially the user-editable portion.

2.5.1.3 Element

A element is used to define information that is stored in a Memory Space (see the OpenLCB Memory Configuration Protocol). Some nodes may have only one memory space, others may have multiple. There can be multiple segments that bind to the same Memory Space.

Node developers are advised to assign relatively short <name> values to segments, because certain Configuration Tools may use these as labels for a tabbed dialog box.

2.5.1.4 Data Elements

This is the point of the whole standard: the actual Variables. The most important information for each variable is how to access it.

The Memory Configuration Protocol requires two values to access a piece of information: the Space number (8-bit integer) and the Address (32-bit integer). The Space value comes from the element.

Addresses are laid out by data length using a depth-first traversal algorithm:

1. At the start of the segment, set current_address to be the value of the attribute `origin' of the element, or 0 (zero) if this attribute is not present.
2. For each child element:
    - increment current_address by the attribute `offset' if present (can be negative!)
    - if it is a primitive type
        - render the UI according to the element type, take current_address for data
        - post-increment current_address by the element size (8 for events, otherwise given by the `size' attribute)
    - if it is a group, check the replication count
        - render a box or separator and header except if group is empty
        - recurse into the group to lay out all child elements
        - if there is any remaining replication left, repeat from to rendering box or separator

Data Elements support constraints for the values:

- Integers have <min> and <max> elements inside. They constrain the range with endpoints inclusive, i.e. <min>5</min> allows value 5 but disallows value 4.
- All elements support a <map> which can be used to constrain the possible values to a fixed list, as well as provide user-readable descriptions to the values. Inside the map, the <property> entry is the one that should be written to the node, and the <value> entry is the one that should be displayed to the user.

Be aware that all data is to be written in big-endian byte order.

2.6 Future Expansion

This section provides guarantees that allow for a Configuration Tool designed for the current version of the standard be able to configure a node designed for a future version of the standard, allowing at least a common set of functionality to operate properly.

The guarantees given in the standard allow the algorithm for tracking the current_address to function properly for future minor versions of the standard. This ensures that all known Data Elements are laid out by the Configuration Tool as intended by the CDI developer. Any unknown Data Elements can be

skipped by the Configuration Tool. It is advised for Configuration Tool developers to identify to the user that an unknown element was found and an update to the software of the Configuration Tool may be required to have proper access to it.

On the other hand, a new major version of the standard may define the XML elements or the layout algorithm in a completely different way that prevents a Configuration Tool designed for an earlier version of the standard for properly following the layout algorithm.

3 Background Information

3.1 Environment of Proposal

3.1.1 Requirements

- Each node needs to 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. This allows a quality configuration experience to happen in a completely self-contained system.
- Users want to configure a node entirely over the OpenLCB, without physical interactions, e.g. pushing buttons.

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 retrieval operations should be state-less and idempotent to simplify software at both ends. This is a preference for the CDI retrieval mechanism, not the CDI itself.
- Multiple independent configuration operations can proceed at the same time. Specifically, multiple devices should be able to retrieve correct configuration description information at the same time. This is a preference for the CDI retrieval mechanism, not the CDI itself.

Design Points

- The “Variables” described here are not exactly the same thing as “Configuration Variables” (CVs) or “Node Variables” (NVs) that are discussed elsewhere, e.g. in DCC documentation. Those are aimed at storage, and so are grouped by address. The “Variables” here are grouped by function. “Long address” might be several DCC CVs, but would be one variable in this proposal. Similarly, DCC CV29 has lots of variables within it, each stored as bits.

- The primary design constraints are complexity and size in the OpenLCB device providing CDI, and complexity and size in the device consuming the CDI.
    - Size and complexity in the providing device is the more important constraint. There are more of those devices, they are cost sensitive, and they may not be upgradeable once delivered.
    - Size and complexity in the CDI-consuming device should also be considered. In particular, code complexity is an issue which must be addressed.

- Secondary constraints are testability of the provided information, scalability of the format, and the convenience and availability of a suitable toolchain.
- There is a physical/logical structure to the configuration which the CDI can and should reflect:
  - The basic OpenLCB unit is a "Node". Nodes provide CDI for their needed configuration information. One possible protocol for that is the Memory Configuration Protocol.
  - A Node can contain zero or more "Producers". Each Producer is independently configured. There is no ordering between separate Producers, but they can be numbered for ease of reference.
  - A Node can contain zero or more "Consumers". Each Consumer is independently configured. There is no ordering between separate Consumers, or between individual Consumers and Producers, but they can be numbered for ease of reference.
  - Each Producer or Consumer can be configured with zero or one Events. A hardware port may have more than one Producer or Consumer (for example two: off and on).
  - Each Event has an Identifier which uniquely defines it.
  - To ensure future growth, there is no required "device", "channel" or other grouping within a node. Those may be present in some node types, and CDI must be able to represent them, but may not require any specific organization. In other words, organizing the configuration is left up to the Node developers. One OpenLCB product may be considered superior to another of equivalent hardware capabilities purely because its configuration is easier to understand and use for the end-user.

3.1.3 Storage

The configuration definition is stored in a hierarchical manner.

I) In what follows:

A "String" must be present; an "Optional String" does not have to be. Strings are in UTF-8.

An "Integer" may be signed; if no sign, it's taken as positive.

A "Map" provides a set of named descriptive values. It contains:

- Name: Optional String, if present required to be unique within enclosing group or node
- Description: Optional String
- 1 or more "Key", "Value" pairs. Each element of the pair can be of any supported type, depending only on how it is to be used.

Map elements provide a mapping between the pairs they contain. For example, a map can relate numeric values for a variable to descriptive strings. A map can also be used to provide free-form documentation when neither the key nor the value are specified in advance. It may be useful in the future to specify how maps can be defined at the group level to reduce duplication. Having the possibility of a “Name” is meant to ease that future effort.

II) At the top, root level is the information for a "node". This includes:

- Manufacturer: String
- Model: If present, the human-readable model name the manufacturer gives to this node.

- Version: If present, the human-readable version string for the current board.
- An optional Map for storing custom attributes defined by the manufacturer. Any other information desired can be added to this map.

III) Within the node is zero or more "groups". Each group contains:

- Name: String, required to be unique within enclosing group or node
- Description: Documents to the end-user the logical contents.
- Replication count: Integer >= 1 (number of times this group is replicated within the parent item)

A group with a replication count > 1 (called a replicated group) can be used to represent a type of replicated device. For example, a node with 4 identical input devices and 6 identical output devices can be compactly described by two groups, with replication counts of 4 and 6 respectively.

Instances within replicated groups are numbered from 1 to the replication count. If more than one replicated group is present, the numbering for each starts again with 1.

Groups may contain one or more inner groups, with the same representation. This may continue to any desired level.

IV) Groups may contain "variable" descriptions of various types.

IV-a) A "variable" description contains:

- Type: represented by the element tag. Can be int, string (UTF-8), eventid. Previously other types were also considered, e.g. "bit", "digit" (an unsigned binary-coded-decimal value), "signed" (a binary value with a sign), "unsigned" (a binary value without a sign), "blob" (arbitrary byte vector).
- Name: String, required to be unique within enclosing group or node
- Max: Integer - the maximum value allowed.
- Min: Integer - the minimum value allowed.
- Size: describes how many bytes are occupied by this variable. Also used for integers to describe the data type size (e.g. byte, word, dword etc).
- Description: Optional String
- Default: optional, for integer type only
- Offset: Optional integer offset to skip before this element in the layout, data is otherwise laid out by length in depth-first order.

A variable may contain zero or one map descriptions. If present, the map represents a mapping between possible values (the "Property" part of the map's pairs) and convenient names for them (the "Value" part of the map's pairs).

Note that the current value of a variable is not considered configuration definition information (see item 1A and 1B in the introduction).

Configuration information must not be packed into variables; each variable must represent one type of information. In particular, the use of individual bits within larger values to pack multiple pieces of information is forbidden; those must be represented as individual variables. (How the information is stored internally is up to the designer of the specific device, and is not restricted; this requirement is about access to the information, not about how it's laid out in physical memory. A possible way to organize internally bit-mapped information is to use an address translation scheme which maps each bit to a byte.)

3.2 Example

The following is not meant to show how configuration definition information would be stored, but what kinds of information would be stored. It's a description of a complex accessory decoder, the Digitrax DS54, modified for use in a Producer-Consumer model.

Hopefully the syntax will be self-explanatory. In any case, it's just for this example, not a proposal of any kind.

```
<?xml version="1.0"?>
<cdi xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://openlcb.org/schema/cdi/1/1/cdi.xsd">

<identification>
<manufacturer>Digitrax</manufacturer>
<model>DS54</model>
<hardwareVersion>2.33</hardwareVersion>
</identification>

<acdi/>

<segment space='251'>
  <name>User Identification</name>
  <description>Lets the user add his own description</description>
  <int size='1'>
    <name>Version</name>
  </int>
  <string size='63'>
    <name>Node Name</name>
  </string>
  <string size='64'>
    <name>Node Description</name>
  </string>
</segment>

<segment space="253">
  <int size="2">
    <name>Address</name>
```

```
    <description>The DCC address of the accessory decoder. This is not needed in OpenLCB operation.</description>
    <min>0</min><max>2044</max>
  </int>

  <group replication="4">
    <name>Channels</name>
    <description>Each channel is one pair of output wires and contains two inputs.</description>
    <repname>Channel</repname>

    <group>
      <name>Turnout output</name>

      <int size="1"><name>Output option</name>
        <default>1</default>
        <map>
        <relation><property>1</property><value>Pulse re-triggerable</value></relation>
        <relation><property>2</property><value>Pulse non-retriggerable</value></relation>
        <relation><property>3</property><value>Static light or slow-motion turnout machine</value></relation>
        <relation><property>4</property><value>Blinking lamp</value></relation>
        </map>
      </int>

      <int size="1"><name>Pulse length</name>
        <description>Specifies the length of the output pulses for the pulsed output options. Ignored when output option is not pulsed.</description>
        <default>0</default>
        <map>
        <relation><property>0</property><value>0.125 sec</value></relation>
        <relation><property>1</property><value>0.25 sec</value></relation>
        <relation><property>2</property><value>0.35 sec</value></relation>
        <relation><property>3</property><value>0.5 sec</value></relation>
        <relation><property>4</property><value>0.625 sec</value></relation>
        <relation><property>5</property><value>0.75 sec</value></relation>
        <relation><property>6</property><value>0.9 sec</value></relation>
        <relation><property>7</property><value>1 sec</value></relation>
        <relation><property>8</property><value>2 sec</value></relation>
        <relation><property>9</property><value>3 sec</value></relation>
        <relation><property>10</property><value>4 sec</value></relation>
        <relation><property>11</property><value>5 sec</value></relation>
        <relation><property>12</property><value>6 sec</value></relation>
        <relation><property>13</property><value>7.5 sec</value></relation>
        <relation><property>14</property><value>10 sec</value></relation>
        <relation><property>15</property><value>12 sec</value></relation>
        </map>
```

```
    </int>

    <eventid>
      <name>Turnout closed</name>
      <description>When this event arrives, a turnout will be moved to the closed position or a lamp
will be switched to ON.</description>
    </eventid>
    <eventid>
      <name>Turnout thrown</name>
      <description>When this event arrives, a turnout will be moved to the thrown position or a lamp
will be switched to OFF.</description>
    </eventid>
  </group>

  <group replication="2"><name>Inputs</name>
    <description>Input 1 is the SWITCH input, Input 2 is the AUX input for this channel</description>
    <repname>Input</repname>
    <eventid>
      <name>Input active</name>
      <description>This event is generated when the input line goes active. Independent from trigger
setup.</description>
    </eventid>
    <eventid>
      <name>Input inactive</name>
      <description>This event is generated when the input line goes inactive. Independent from trigger
setup.</description>
    </eventid>

    <group><name>Trigger</name>
      <description>Advanced triggering and local actions</description>

      <int size="1">
        <name>Trigger condition</name>
        <map>
          <relation><property>0</property><value>Positive Edge: OFF to ON</value></relation>
          <relation><property>8</property><value>Negative Edge: ON to OFF</value></relation>
          <relation><property>1</property><value>Positive Level: ON</value></relation>
          <relation><property>9</property><value>Negative Level: OFF</value></relation>
          <relation><property>2</property><value>Output following: Output ON -- Positive
Edge</value></relation>
          <relation><property>3</property><value>Reverse output following: Output ON -- Negative
Edge</value></relation>
        </map>
      </int>
      <eventid>
        <name>Trigger event</name>
```

```
        <description>If set, this event is generated when triggered</description>
      </eventid>
      <int size="1">
        <name>Action</name>
        <default>1</default>
        <map>
          <relation><property>1</property><value>No Action</value></relation>
          <relation><property>0</property><value>Output toggle</value></relation>
          <relation><property>2</property><value>Output Thrown</value></relation>
          <relation><property>3</property><value>Output Closed</value></relation>
          <relation><property>7</property><value>Output Follows Input</value></relation>
          <relation><property>4</property><value>Turntable Pause</value></relation>
          <relation><property>5</property><value>Turntable Continue</value></relation>
          <relation><property>6</property><value>Local Route</value></relation>
        </map>
      </int>
    </group>
  </group>

  <int size="1"><name>Generate output events</name>
    <description>If set, local output modification by Advanced triggers will generate the same events
that would trigger that output change.</description>
    <default>0</default>
    <map>
      <relation><property>0</property><value>off</value></relation>
      <relation><property>1</property><value>on</value></relation>
    </map>
  </int>
 </group>
</segment>
</cdi>
```

This example tries to follow the configuration options of the DS54 moderately accurately. This means that most of the features are present, and where meaningful, the enumeration values were kept, but in a number of cases the conceptual model that the DS54 is following (LocoNet device and DCC accessory decoder) does not fit the conceptual model of an OpenLCB node. These conceptual differences were incorporated in some cases, but there was no attempt made on redesigning the product into a great OpenLCB Accessory Decoder, since that is not subject of this Technical Note.

- Addressing
  - The CV addressing scheme is not kept, because the address space of OpenLCB Configuration Space is not limited in size like that of DCC CVs using handheld throttles as configuration tools.
  - In particular there is no reason to encode multiple options in one CV, so the upper nibble – lower nibble encoding that the Triggers and Actions use was not reproduced. The

conceptual requirement for the Variables in CDI is that each variable must be independently addressable, and byte is the smallest addressing unit.
    - No attempt was made to make the address layout friendly to the decoder code. In real life one might want to use the offset feature to lay out variables at aligned boundaries and repeated groups to have some nice alignment (say each repetition is 32 bytes) to allow for easier pointer manipulation.
- In a real DS54, there are subtle differences between the Switch and Aux configuration choices on the various channels. Here we just used replication. For a real device, they could either be separately specified or (more likely) the differences wouldn't matter in a P/C-based device. Also for a product choice the inputs would probably be better as numbered than named.
- Commands
    - Instead of receiving DCC turnout commands, the Output entities receive a pair of events to set the output to thrown or closed. There are alternate design options here: since each output channel is actually a pair of independently controllable outputs, there are four actual states (off-off, off-on, on-off, on-on). To reach all four states, a real device might want to add additional consumers and their configurable Event IDs. Also when receiving the DCC turnout commands, the four states interact with the “Output Type” setting in weird and wonderful ways. This was not represented in the example above.
- Inputs
    - The input lines, symmetrically to the output lines, use a pair of Event IDs to generate native OpenLCB commands. For most of the use-cases this makes the Advanced Triggering feature unnecessary. For example the input's native Event ID can be configured to be the same as the output thrown event, which will establish both the necessary local control as well as the reporting of local control (see “Generate output events” setting). In fact this is even more powerful, since local control is not restricted to the same output channel as the input button is assigned to.
    - Nevertheless, the Triggering concept was mapped into this configuration, and the Trigger can also be observed on the OpenLCB bus if needed by programming an Event ID into the Trigger producer. In a real world the user would probably either use the Active+Inactive producers or the Trigger producer, but probably not both.
- Loconet messages
    - The original DS54 has extensive settings for generating various LocoNet messages. These are all superfluous, since the OpenLCB Event concept subsumes all the different use-cases (including power off, turnout feedback, sensor input, etc. -- they are all just Events)
- Local routes and turnout chaining
    - In the Producer/Consumer model in OpenLCB these features are unnecessary, because the control model natively supports throwing multiple turnouts by a single Event on the bus. Just configure the same Event ID into all the respective turnouts for the desired direction. It is also possible to cause the feedback message from one input to throw another turnout by configuring them for the same Event ID.

3.3 Notes about compression

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

Some references for XML compression:

http://www.ibm.com/developerworks/xml/library/x-datacompression/index.html?cmp=dw&cpb=dwxml&ct=dwnew&cr=dwnen&ccy=zz&csr=072111

http://www.cs.panam.edu/~artem/main/teaching/csci6370spring2011/papers/XML%20compression%20techniques%20A%20survey%20and%20comparison.pdf

On the other hand, a look-back compression algorithm has the advantage that it's cheap to decompress and might do almost as well:

http://excamera.com/sphinx/article-compression.html

XML strings can start with a UTF BOM (either 0xEF, 0xFF or 0xFE in the 1st byte, since there's no need to support UTF-32BE or UTF-32LE), or the UTF-8 text for "<?xml" which starts with 0x3C. (But concern about support for too many character sets!)

A first byte of 0x80 is defined as the "Compressed" indicator(s), followed by a byte that indicates the compression type. (We don't want to have too many kinds, as receivers need to implement all of them to be able to use the CDI!) 0x80 00 is the code for our choice of default, see http://excamera.com/sphinx/article-compression.html

Table of Contents

1 Introduction....1
2 Annotations to the Standard....1
2.1 Introduction....1
2.2 Intended Use....1
2.3 Reference and Context....1
2.4 Content....2
2.5 Format....2
2.5.1 XML Elements....2
2.5.1.1 <identification> Element....2
2.5.1.2 <acdi> Element....3
2.5.1.3 Element....3
2.5.1.4 Data Elements....4
2.6 Future Expansion....4
3 Background Information....5
3.1 Environment of Proposal....5
3.1.1 Requirements....5
3.1.2 Preferences....5
Design Points....5
3.1.3 Storage....6
3.2 Example....8
3.3 Notes about compression....13