OpenLcbCLib

Developer Guide

From first project to a fully featured OpenLCB/LCC node

Covers all supported platforms and IDEs

Table of Contents

1. Introduction

OpenLcbCLib is a C library that implements the OpenLCB/LCC (Layout Command Control) protocol suite for microcontrollers. It is designed to be portable across any hardware platform and IDE, with no dynamic memory allocation — all buffers are statically sized at compile time, making it suitable for processors with very limited RAM.

This guide walks you through creating a complete node project from scratch using the Node Wizard tool, then explains every generated file so you understand exactly what the code does and how to extend it.

What the Library Does (and Does Not Do)

The library handles all OpenLCB/LCC protocol compliance: alias negotiation, node initialization, event transport, datagram handling, SNIP, configuration memory access, and train control. It does NOT include any hardware-specific code. You supply short driver functions that read and write your CAN hardware, and the library calls them as needed.

Platform Support

Because the library contains no hardware-specific code, it can be ported to any processor and toolchain. You provide a small set of driver functions for your CAN hardware and platform services, and the library does the rest. The following processors already have driver code written and example projects included in the library:

Processor / Board	Example IDE / Toolchain
ESP32	Arduino IDE, PlatformIO
Raspberry Pi Pico (RP2040)	Arduino IDE, PlatformIO
STM32 (F4 and others)	STM32 Cube IDE
Microchip dsPIC33	MPLAB X
TI MSPM0	Code Composer Studio
macOS (simulation)	Xcode

Porting to a new processor means writing two short driver files (CAN hardware and platform services). The Node Wizard generates TODO-marked stubs for both so you know exactly what functions need to be filled in. Section 7 covers this in detail.

This guide focuses on ESP32 as the worked example. All concepts apply equally to any other target.

2. OpenLCB/LCC Concepts

You do not need to read the full OpenLCB/LCC standards to use this library, but a basic understanding of the following concepts will help you make good design decisions.

The Network

All LCC nodes connect to a shared CAN bus (or a TCP/IP gateway). Every node can hear every message on the bus. Nodes identify themselves with a globally unique 48-bit Node ID.

Events (Producer-Consumer Model)

The most common way nodes communicate is through Events. An event is a 64-bit number that means "something happened." A node that detects a condition (button pressed, train entered block) fires an event. Any node that cares about that event reacts to it (throws a switch, changes a signal). The producer and consumer do not know about each other — they are loosely coupled through the shared event ID.

Configuration (CDI)

A Typical node exposes configurable settings to the layout operator via CDI (Configuration Description Information). The CDI is an XML document stored in the node that describes what settings are available. JMRI and other tools read the CDI and show a configuration panel.

Node Types and What They Compile

Node Type	Protocols Compiled	Typical Use
Basic	Events only	Simple sensor or output that only fires/reacts to events.
Typical	Events + Datagrams + Config Memory	Most common. Sensor/output with user-configurable event IDs.
Train	Typical + Train + FDI	Locomotive decoder. Responds to speed/function commands.
Train Controller	Typical + Train + Train Search	Throttle or command station. Commands trains.

3. Project File Structure

Understanding the file layout tells you where to look when you need to change something. The files in src/openlcb/ and src/drivers/ are library code — do not edit them. The files at the top level are yours.

```
YourProject/                        <- Arduino sketch folder
  YourProject.ino                   <- Main sketch (setup + loop)
  openlcb_user_config.h             <- REQUIRED: buffer sizes & feature flags
  openlcb_user_config.c             <- User node parameter data
  callbacks.h / callbacks.c         <- Your application logic hooks
  esp32_drivers.h / .c              <- Platform services (NVS, mutex, reboot)
  esp32_can_drivers.h / .c          <- CAN hardware interface
  src/
    openlcb/                        <- Library core (do not edit)
      openlcb_config.h/c            <- Stack init API
      openlcb_types.h               <- All data types
      openlcb_node.h/c              <- Node allocation and management
      openlcb_application.h/c       <- Event registration/sending API
      protocol_event_transport.h/c <- Event protocol engine
      protocol_datagram_handler.h/c<- Datagram protocol engine
      protocol_config_mem_*.h/c     <- Config memory protocol
      protocol_snip.h/c             <- Node identification protocol
    drivers/
      canbus/                       <- CAN state machines (do not edit)
```

4. The Node Wizard in Detail

The Node Wizard is a browser-based code generator at tools/node_wizard/node_wizard.html. Open it in any modern browser — no installation or internet connection needed. Work through the sidebar sections from top to bottom. All settings persist automatically in your browser's local storage. Use **Save Project** / **Load Project** in the header to save and reload a project across browsers or machines.

4.1 Node Type and Project Options

The first step is selecting the role of your node. This determines which feature flags are set in openlcb_user_config.h, which sidebar sections are enabled, and which callback stubs are generated.

Node Type	Description
Basic	Sends and receives events only. No configurable settings. CDI and FDI sections are disabled.
Typical	Events plus user-configurable settings stored in config memory. The CDI section is enabled. Most common choice for sensors, turnout drivers, and signal heads.
Train	Locomotive decoder. CDI and FDI sections are both enabled.
Train Controller	Throttle or command station.
Custom	Advanced — coming soon.

Project Options

Below the node type buttons, the **Project Options** panel is always visible. Fill this in before generating — the values flow into the generated code and the ZIP filename.

Field	Description
Project Name	Short identifier for your project, e.g. my_signal_controller. Used as the ZIP filename when you download. Stick to letters, numbers, and underscores.
Author	Your name or organisation. Written into the generated file headers.
Node ID	Your node's globally unique 48-bit identifier in dotted hex format: xx.xx.xx.xx.xx.xx (e.g. 05.01.01.00.00.01). The field auto-formats as you type and shows a live valid / invalid indicator. NMRA members may use their assigned member range; others may request a range at registry.openlcb.org/uniqueidranges. For private testing use any value starting with 05.01.01.
Broadcast Time	Optionally compile in the OpenLCB Broadcast Time protocol. Choose None (default), Producer (this node drives the layout clock), or Consumer (this node follows the layout clock).
Firmware Update	Enables the OpenLCB Firmware Upgrade Protocol so the node can accept a new firmware image over the bus without physical access. Requires config memory — disabled for Basic nodes.

4.2 CDI Editor

The CDI editor lets you describe what settings your node exposes to configuration tools like JMRI. The CDI is an XML document. The editor provides a visual interface so you do not need to write raw XML. This section is enabled for Typical, Train, and Train Controller nodes.

CDI Structure

A CDI document has two main parts: an identification block (manufacturer, model, versions) and one or more configuration segments. A segment is a named block of settings stored at a specific memory address.

```
<?xml version="1.0"?>
<cdi>
  <identification>
    <manufacturer>Your Name</manufacturer>
    <model>My Sensor Node</model>
    <hardwareVersion>1.0</hardwareVersion>
    <softwareVersion>1.0</softwareVersion>
  </identification>

  <!-- Standard: user name and description at address 0, space 253 -->
  <segment origin="0" space="253">
    <name>Node</name>
    <group>
      <string size="62"><name>Node Name</name></string>
      <string size="63"><name>Node Description</name></string>
    </group>
  </segment>

  <!-- Application segment: your configurable event IDs -->
  <segment origin="128" space="253">
    <name>Sensor</name>
    <group replication="4">
      <repname>Input </repname>
      <eventid><name>Active Event</name></eventid>
      <eventid><name>Inactive Event</name></eventid>
    </group>
  </segment>
</cdi>
```

CDI Element	Description
<string size="N">	Text field N bytes long. Used for names and descriptions.
<eventid>	Stores a 64-bit event ID (8 bytes). Most common field type.
<int size="N">	Integer value N bytes (1, 2, or 4).
<group replication="N">	Repeats the group N times. Useful for multi-channel nodes.
	Space 253 is the user configuration space JMRI reads/writes.

Important: The total bytes in your CDI segments must match the actual memory layout in your config_mem_read/write driver functions. Mismatches cause JMRI to read garbage values.

4.3 FDI Editor (Train Nodes Only)

The FDI editor is only enabled for Train nodes. It describes what DCC functions your decoder supports (headlight, bell, horn, etc.). Each function entry maps a function number (F0-F28) to a name and behaviour type (binary on/off, or momentary).

4.4 Platform Drivers

The Platform Drivers section is where you select your target hardware. This is one of the most important steps — the platform you choose determines two things: whether the output uses Arduino or

non-Arduino layout, and whether the driver files come pre-filled with working code or just TODO stubs.

Platform	Framework	Layout	Driver Code
ESP32 + TWAI	Arduino / PlatformIO	Arduino	Pre-filled
ESP32 + WiFi GridConnect	Arduino / PlatformIO	Arduino	Pre-filled
RP2040 + MCP2517FD	Arduino (Earle Philhower core)	Arduino	Pre-filled
STM32F4xx + CAN	STM32 HAL (CubeIDE)	Non-Arduino	Pre-filled
TI MSPM0 + MCAN	TI DriverLib (CCS / Theia)	Non-Arduino	Pre-filled
None / Custom	Your own toolchain	Non-Arduino	TODO stubs

Each platform also exposes configurable parameters — for example, GPIO pin numbers for the CAN transceiver on ESP32, or SPI pin assignments for the RP2040. These are substituted into the generated driver code.

4.5 Callbacks

Callbacks are the hooks between the library and your application. The Callbacks section lets you choose which protocol events your application needs to respond to. Each selected callback generates a stub with a comment explaining what it should do. Unselected callbacks are omitted entirely, keeping the code clean.

4.6 Generated Files

Click the **Generated Files** tile in the sidebar to preview the output file tree. From there, click **Download ZIP** to get your project. The folder layout and file extensions inside the ZIP depend on the platform selected in step 4.4.

Generated File	What It Is
openlcb_user_config.h	Buffer depths and feature flags. Required.
openlcb_user_config.c	The node_parameters struct with your node's metadata.
main.ino (Arduino)	Application entry point — .ino so Arduino IDE recognises it.
main.c (non-Arduino)	Application entry point — plain C for other toolchains.
application_callbacks/callbacks_*.h	Callback header stubs — same in both modes.
callbacks_*.cpp (Arduino)	Callback implementations — .cpp so C++ Arduino APIs (Serial, Wire, etc.) are available.
callbacks_*.c (non-Arduino)	Callback implementations — plain C.
application_drivers/openlcb_can_drivers.h	CAN hardware abstraction header.
openlcb_can_drivers.cpp / .c	CAN driver implementation — .cpp in Arduino mode, .c otherwise.
application_drivers/openlcb_drivers.h	Platform driver header (lock/unlock, config read/write, reboot).
openlcb_drivers.cpp / .c	Platform driver implementation — .cpp in Arduino mode, .c otherwise.
xml_files/cdi.xml	Your CDI XML definition.
xml_files/fdi.xml	Your FDI XML (train nodes only).
GETTING_STARTED.txt	Step-by-step instructions included in every ZIP.
_project.json	Saved Wizard state. Reload it to resume editing.

4.7 Arduino vs. Non-Arduino Differences

The platform you select in the **Platform Drivers** step automatically determines the output layout. There is no manual checkbox — the Wizard sets Arduino mode based on the chosen platform. This changes two things in the generated output:

- **Folder structure** — all driver, callback, library, and XML folders move under **src/**. Arduino IDE only compiles C/C++ files that are in a subfolder named exactly src/.
- **File extensions** — driver and callback source files are generated as **.cpp** instead of .c. This lets you call C++ Arduino APIs such as Serial, Wire, and SPI directly inside your driver and callback implementations.

Arduino Layout (ESP32, RP2040)

```
<project>/                              <- sketch folder (same name as .ino)
|-- main.ino                            <- application entry point
|-- openlcb_user_config.h               <- feature flags and buffer sizes
|-- openlcb_user_config.c               <- node parameters struct
|-- src/                                <- Arduino IDE compiles this subtree
|   |-- application_drivers/
|   |   |-- openlcb_can_drivers.h       <- CAN hardware interface
|   |   |-- openlcb_can_drivers.cpp
|   |   |-- openlcb_drivers.h           <- platform services
|   |   `-- openlcb_drivers.cpp
|   |-- application_callbacks/
|   |   |-- callbacks_*.h               <- your application logic
|   |   `-- callbacks_*.cpp
|   |-- xml_files/
|   |   |-- cdi.xml
|   |   `-- fdi.xml                     <- train nodes only
|   `-- openlcb_c_lib/                  <- copy library here
|       |-- openlcb/
|       |-- drivers/canbus/
|       `-- utilities/
|-- GETTING_STARTED.txt
`-- <type>_project.json                 <- reload in Wizard to resume editing
```

Non-Arduino Layout (STM32, MSPM0, None/Custom)

```
<project>/                              <- project root
|-- main.c                              <- application entry point
|-- openlcb_user_config.h               <- feature flags and buffer sizes
|-- openlcb_user_config.c               <- node parameters struct
|-- application_drivers/
|   |-- openlcb_can_drivers.h           <- CAN hardware interface
|   |-- openlcb_can_drivers.c
|   |-- openlcb_drivers.h               <- platform services
|   `-- openlcb_drivers.c
|-- application_callbacks/
|   |-- callbacks_*.h                   <- your application logic
|   `-- callbacks_*.c
|-- xml_files/
|   |-- cdi.xml
|   `-- fdi.xml                         <- train nodes only
|-- openlcb_c_lib/                      <- copy library here
|   |-- openlcb/
|   |-- drivers/canbus/
|   `-- utilities/
|-- GETTING_STARTED.txt
`-- <type>_project.json                 <- reload in Wizard to resume editing
```

Note: The openlcb_c_lib/ folder is a placeholder — the Wizard creates the empty folder structure but does not copy the library. You must copy the openlcb/, drivers/canbus/, and utilities/ source files there yourself.

5. openlcb_user_config.h in Depth

This file is the single most important file in your project. The library uses these constants at compile time to determine how much memory to allocate. If a value is wrong, the library will either waste RAM or crash at runtime.

5.1 Feature Flags

```
// Uncomment the protocols your node uses.
// Unused protocols are excluded from the build entirely.

#define OPENLCB_COMPILE_EVENTS           // Event producer/consumer
#define OPENLCB_COMPILE_DATAGRAMS        // Required for Config Memory
#define OPENLCB_COMPILE_CONFIG_MEMORY    // CDI and settings via JMRI
// #define OPENLCB_COMPILE_BROADCAST_TIME   // Clock synchronization
// #define OPENLCB_COMPILE_TRAIN            // Locomotive control
// #define OPENLCB_COMPILE_TRAIN_SEARCH     // Throttle train discovery
```

Note: If you enable OPENLCB_COMPILE_CONFIG_MEMORY you must also enable OPENLCB_COMPILE_DATAGRAMS. Config memory uses datagrams as its transport layer.

5.2 Message Buffer Pool

The library uses a pool of fixed-size message buffers. Choose depths that balance RAM use with the traffic your node will handle.

Buffer Type (define name)	Size per Buffer
USER_DEFINED_BASIC_BUFFER_DEPTH	16 bytes each — most OpenLCB messages fit here
USER_DEFINED_DATAGRAM_BUFFER_DEPTH	72 bytes each — datagram protocol messages
USER_DEFINED_SNIP_BUFFER_DEPTH	256 bytes each — SNIP replies and payload events
USER_DEFINED_STREAM_BUFFER_DEPTH	512 bytes each — stream data transfer

5.3 Node Count

USER_DEFINED_NODE_BUFFER_DEPTH controls how many virtual nodes this device can host. A simple accessory decoder uses 1. A command station managing multiple locomotives may use one slot per active locomotive.

5.4 Event Counts

```
#define USER_DEFINED_PRODUCER_COUNT       64  // max events this node can produce
#define USER_DEFINED_CONSUMER_COUNT       32  // max events this node can react to
#define USER_DEFINED_PRODUCER_RANGE_COUNT  5  // must be >= 1
#define USER_DEFINED_CONSUMER_RANGE_COUNT  5  // must be >= 1
```

5.5 CDI and FDI Buffer Sizes

```
// These must be large enough to hold the full XML text of your CDI/FDI.
#define USER_DEFINED_CDI_LENGTH   20000  // bytes
#define USER_DEFINED_FDI_LENGTH    1000  // bytes (set small if not a train)
```

---

6. Initialization — main.ino

The main sketch file wires together the hardware drivers, the library stack, and your application callbacks. The pattern is always the same four steps: define the config structs, initialize hardware, initialize the library, create your node(s).

6.1 The CAN Config Struct

This struct tells the CAN state machine how to talk to your CAN hardware. All fields are function pointers. The library calls these when it needs to send a frame, check the TX buffer, or notify you of incoming frames and alias changes.

```
static const can_config_t can_config = {
    // How to send one raw CAN frame to the hardware
    .transmit_raw_can_frame  = &Esp32CanDriver_transmit_raw_can_frame,

    // Returns true if the CAN TX buffer is ready for another frame
    .is_tx_buffer_clear      = &Esp32CanDriver_is_can_tx_buffer_clear,

    // Mutex for thread safety (FreeRTOS on ESP32)
    .lock_shared_resources   = &Esp32Drivers_lock_shared_resources,
    .unlock_shared_resources = &Esp32Drivers_unlock_shared_resources,

    // Called when a CAN frame arrives or departs
    .on_rx             = &Callbacks_on_can_rx_callback,
    .on_tx             = &Callbacks_on_can_tx_callback,

    // Called when our node alias changes (rare - usually only at startup)
    .on_alias_change   = &Callbacks_alias_change_callback,
};
```

6.2 The OpenLCB Config Struct

This struct tells the OpenLCB protocol engine how to interact with your application. The most important entries are the config memory read/write functions and the reboot function.

```
static const openlcb_config_t openlcb_config = {
    .lock_shared_resources   = &Esp32Drivers_lock_shared_resources,
    .unlock_shared_resources = &Esp32Drivers_unlock_shared_resources,

    // Read/write your node's non-volatile settings memory
    .config_mem_read         = &Esp32Drivers_config_mem_read,
    .config_mem_write        = &Esp32Drivers_config_mem_write,

    // Restart the microcontroller (called after firmware update)
    .reboot                  = &Esp32Drivers_reboot,

    // Application callbacks for protocol events
    .factory_reset           = &Callbacks_operations_request_factory_reset,
    .freeze                  = &Callbacks_freeze,
    .unfreeze                = &Callbacks_unfreeze,
    .firmware_write          = &Callbacks_write_firmware,

    // Called every 100ms by the library timer tick
    .on_100ms_timer          = &Callbacks_on_100ms_timer_callback,
};
```

6.3 Setup and Loop

```
void setup() {
    Serial.begin(9600);

    // 1. Initialize hardware
    Esp32CanDriver_setup();
    Esp32Drivers_setup();

    // 2. Initialize the library stacks
    CanConfig_initialize(&can_config);
    OpenLcb_initialize(&openlcb_config);

    // 3. Initialize your application callbacks
    Callbacks_initialize();

    // 4. Create the node
    OpenLcb_create_node(NODE_ID, &OpenLcbUserConfig_node_parameters);
}

void loop() {
    // Run all state machines. Call as fast as possible.
    OpenLcb_run();

    // Your application code here. Keep it short and non-blocking.
}
```

Note: OpenLcb_run() must be called repeatedly from your main loop. Do not use delay() in loop() — it stalls the state machines.

7. Implementing the Drivers

The two driver files are the only hardware-specific code in a project. If you are using the ESP32 BasicNode example, these are already implemented. This section explains what each function must do when porting to a new platform.

7.1 CAN Bus Drivers (esp32_can_drivers.c)

transmit_raw_can_frame

Called by the library to send one CAN frame. The library will not call this again until is_tx_buffer_clear() returns true.

```
void Esp32CanDriver_transmit_raw_can_frame(can_msg_t *can_msg) {
    twai_message_t msg;
    msg.extd             = 1;   // OpenLCB uses 29-bit extended IDs
    msg.identifier       = can_msg->id;
    msg.data_length_code = can_msg->dlc;
    for (int i = 0; i < can_msg->dlc; i++)
        msg.data[i] = can_msg->data[i];
    twai_transmit(&msg, 0);     // non-blocking
}
```

is_tx_buffer_clear

```
bool Esp32CanDriver_is_can_tx_buffer_clear(void) {
    twai_status_info_t info;
    twai_get_status_info(&info);
    return (info.msgs_to_tx == 0);
}
```

Receiving Frames

Incoming CAN frames are not pulled by the library. Your setup code creates a FreeRTOS task (or polling loop) that reads frames from the hardware and passes them into the library:

```
void can_receive_task(void *param) {
    twai_message_t msg;
    for (;;) {
        if (twai_receive(&msg, pdMS_TO_TICKS(10)) == ESP_OK) {
            can_msg_t can_msg;
            can_msg.id  = msg.identifier;
            can_msg.dlc = msg.data_length_code;
            for (int i = 0; i < msg.data_length_code; i++)
                can_msg.data[i] = msg.data[i];
            CanMainStateMachine_process_rx_frame(&can_msg);
        }
    }
}
```

7.2 OpenLCB Platform Drivers (esp32_drivers.c)

Configuration Memory Read and Write

The library calls these when JMRI reads or writes your node's settings. You map addresses to your NVS or EEPROM.

```
uint16_t Esp32Drivers_config_mem_read(
    openlcb_node_t *node, uint32_t address,
    uint16_t count, uint8_t *buffer)
{
    preferences.getBytes("cfg", scratch_buf, sizeof(scratch_buf));
    memcpy(buffer, scratch_buf + address, count);
    return count;
}

uint16_t Esp32Drivers_config_mem_write(
    openlcb_node_t *node, uint32_t address,
    uint16_t count, uint8_t *buffer)
{
    preferences.getBytes("cfg", scratch_buf, sizeof(scratch_buf));
    memcpy(scratch_buf + address, buffer, count);
    preferences.putBytes("cfg", scratch_buf, sizeof(scratch_buf));
    return count;
}
```

Mutex and Reboot

```
void Esp32Drivers_lock_shared_resources(void) {
    xSemaphoreTake(mutex_handle, portMAX_DELAY);
}
void Esp32Drivers_unlock_shared_resources(void) {
    xSemaphoreGive(mutex_handle);
}
void Esp32Drivers_reboot(void) {
    esp_restart();
}
```

---

8. Callbacks — Where Your Application Lives

Callbacks are function stubs the library calls when protocol events occur. This is where you write your application logic. The Node Wizard generates the function signatures with TODO comments. You fill in the bodies.

Callbacks_initialize()

Called once during setup, after OpenLcb_initialize(). Use this to register event IDs, load settings from NVS, and initialize output pins.

```
void Callbacks_initialize(void) {
    openlcb_node_t *node = OpenLcb_get_node(0);

    // Register events this node produces
    OpenLcbApplication_register_producer_eventid(
        node, 0x0501010107770001, EVENT_STATUS_CLEAR);

    // Register events this node consumes
    OpenLcbApplication_register_consumer_eventid(
        node, 0x0501010107770002, EVENT_STATUS_UNKNOWN);

    pinMode(BUTTON_PIN, INPUT_PULLUP);
    pinMode(LED_PIN, OUTPUT);
}
```

Callbacks_on_100ms_timer_callback()

The library calls this function every 100 milliseconds. Good place for polling inputs and debouncing buttons.

```
static uint8_t button_prev = HIGH;

void Callbacks_on_100ms_timer_callback(void) {
    openlcb_node_t *node = OpenLcb_get_node(0);
    uint8_t button_now = digitalRead(BUTTON_PIN);
    if (button_now == LOW && button_prev == HIGH)
        OpenLcbApplication_send_event_pc_report(
            node, 0x0501010107770001);
    button_prev = button_now;
}
```

Callbacks_operations_request_factory_reset()

Called when a user triggers a factory reset from a configuration tool. Erase all stored settings and restore defaults.

Callbacks_freeze() and Callbacks_unfreeze()

Called before and after a firmware update. Freeze stops all application activity. Unfreeze restores normal operation.

CAN Observation Callbacks (Optional)

Callbacks_on_can_rx_callback and Callbacks_on_can_tx_callback let you observe every raw CAN frame. Most applications leave these as empty stubs. Useful for debugging.

---

9. Events in Depth

Choosing Event IDs

An Event ID is a 64-bit number. The convention is: the upper 6 bytes come from your Node ID, and the lower 2 bytes are a sequence number you choose. This ensures your event IDs are globally unique.

```
// Pattern: Node ID bytes + your own sequence number
// Node ID:  0x05.01.01.01.07.77
#define EVENT_BUTTON_PRESSED   0x0501010107770001
#define EVENT_BUTTON_RELEASED  0x0501010107770002
#define EVENT_SENSOR_ACTIVE    0x0501010107770003
#define EVENT_SENSOR_INACTIVE  0x0501010107770004
```

Event Status Values

Status	Meaning
EVENT_STATUS_CLEAR	The condition this event represents is currently inactive/off.
EVENT_STATUS_SET	The condition is currently active/on.
EVENT_STATUS_UNKNOWN	Current state is unknown (most common for consumers at startup).

Loading Event IDs from Configuration Memory

In a configurable node, event IDs are stored in NVS so the user can change them through JMRI. Load them in Callbacks_initialize() before registering.

```
static uint64_t event_active = 0x0501010107770001; // default

void Callbacks_initialize(void) {
    openlcb_node_t *node = OpenLcb_get_node(0);

    // Load from NVS - address must match your CDI layout
    Esp32Drivers_config_mem_read(
        node, 128, 8, (uint8_t*)&event_active);

    OpenLcbApplication_register_producer_eventid(
        node, event_active, EVENT_STATUS_UNKNOWN);
}
```

Note: Event IDs in configuration memory are big-endian (most significant byte first). On little-endian processors like ESP32, you may need to byte-swap them after reading.

10. Configuration Memory

When OPENLCB_COMPILE_CONFIG_MEMORY is enabled, configuration tools like JMRI can read and write your node's settings remotely. The settings are stored as a flat byte array in non-volatile

storage. The CDI tells JMRI how to interpret the bytes.

Memory Space Layout

Space	Purpose
253 (0xFD)	User configuration. CDI describes this space. JMRI reads and writes here.
252 (0xFC)	All memory (read only). Used internally by the library.
251 (0xFB)	CDI XML. The library serves CDI from here.
249 (0xF9)	FDI XML (train nodes only).

Designing Your Memory Layout

The bytes in space 253 are laid out exactly as described in your CDI segments. Plan the layout before writing CDI:

```
// Example layout for a 4-channel sensor node:
//
// Address  Size  Content
// -------  ----  --------------------------------
//   0       62   User node name (string)
//  62       63   User node description (string)
// 128        8   Channel 1 Active Event ID
// 136        8   Channel 1 Inactive Event ID
// 144        8   Channel 2 Active Event ID
// 152        8   Channel 2 Inactive Event ID
// 160        8   Channel 3 Active Event ID
// 168        8   Channel 3 Inactive Event ID
// 176        8   Channel 4 Active Event ID
// 184        8   Channel 4 Inactive Event ID
// Total: 192 bytes
```

The CDI XML to C Array Tool

The CDI XML must be embedded in the firmware as a byte array. The library includes a tool at tools/cdi_to_array/ that converts your node.cdi file into a C header:

```
# Run the converter:
python tools/cdi_to_array/cdi_to_array.py node.cdi cdi_array.h

// Output defines:
// const uint8_t CDI_ARRAY[] = { 0x3C, 0x3F, ... };
// const uint16_t CDI_ARRAY_LEN = 1234;
```

11. Train Nodes (Brief Overview)

Select Train or Train Controller in the Node Wizard to add train-specific protocols.

Train Node

A Train node represents a single locomotive and accepts speed and function commands from throttles over the LCC network.

```
// Called when a throttle sends a speed command:
void Callbacks_train_set_speed(
    openlcb_node_t *node, float speed, bool forward)
{
    my_dcc_driver_set_speed(speed, forward);
}

// Called when a throttle presses a function button (F0-F28):
void Callbacks_train_set_function(
    openlcb_node_t *node, uint8_t fn, bool active)
{
    my_dcc_driver_set_function(fn, active);
}
```

Train Controller

A Train Controller (throttle or command station) discovers and commands trains on the network using the Train Search protocol. This node type does not have an FDI.

---

12. Using PlatformIO Instead of Arduino IDE

PlatformIO in VSCode is an alternative to Arduino IDE. Here is a minimal platformio.ini for the BasicNode example:

```
[env:esp32dev]
platform    = espressif32
board       = esp32dev
framework   = arduino

; Add the sketch directory to the include path
build_flags = -I src
```

Place all source files in the src/ directory. PlatformIO will compile all .c and .cpp files in src/ automatically.

---

13. Troubleshooting

Symptom	Likely Cause and Fix
Node does not appear in JMRI	Check CAN wiring (TX/RX may be swapped). Verify 120-ohm termination resistors on bus. Check NODE_ID is unique.
Undefined reference to USER_DEFINED_*	openlcb_user_config.h is not on the include path. For Arduino IDE it must be in the sketch folder. For PlatformIO add -I src.
Node appears then immediately disappears	Another node has the same Node ID. Every node must have a unique ID.
JMRI config panel is empty	CDI is malformed or the CDI byte array was not regenerated after editing the XML. Re-run the cdi_to_array tool.
Settings do not save across power cycles	config_mem_write is not writing to NVS. Add Serial.println() in the write function to confirm it is being called.
Events not received by consumer	Verify both nodes are on the same bus. Verify the consumer registered the exact same 64-bit event ID as the producer.
Buffer depth compile error on 8-bit processors	Total buffer count (sum of all four depth defines) must not exceed 126 on 8-bit platforms.

Enabling Verbose Compilation Output

```
// In openlcb_user_config.h:
#define OPENLCB_COMPILE_VERBOSE
// Prints a feature summary confirming which protocols were compiled in.
```

Online Resources

Library documentation: https://jimkueneman.github.io/OpenLcbCLib/

OpenLCB/LCC standards: https://www.nmra.org/lcc