This document outlines the coding style used in the OpenLcbCLib library.

Use of types:

This library uses typedef liberally to allow better type checking does not encourage the use of type casting unless absolutely necessary.

File Names and Folders:

Filenames are all lower case with underscores between words and are descriptive. The file structure is as follows

-src
- drivers // This is where drivers are located that take the physical
// layer IO and converts it to raw openlcb message types for
// use in the openlcb core library
- common // This is where the can interface files live to support another transport layer
// (such as TCP/IP) create another folder at this same lever and name it something
// more appropriate.
- openlcb // This where core files that function on the openlcb messages in using full NodeIDs

C Coding Style:

Header Files:

Header file functions use the name of the file in CaMeL case appended with a descriptive name of what the function does in lower case with words separated by underscores. For instance in the c*an_buffer_store.h* file to allocate a new CAN buffer you would call

```
extern can_msg_t* CanBufferStore_allocate_buffer(void);
```

Guards are named with 2 leading and trailing underscores with the name of the module in capitals with underscores between the words:

```
// This is a guard condition so that contents of this file are not included
// more than once.
#ifndef __CAN_BUFFER_STORE__
#define __CAN_BUFFER_STORE__

#ifdef  __cplusplus
extern "C" {
#endif /* __cplusplus */


#ifdef  __cplusplus
}
```

```
#endif /* __cplusplus */

#endif  /* __CAN_BUFFER_STORE__ */
```

Type Definitions:

Any constant that can not be changed is written in capitals with underscores between the letter this includes defines:

```
#define USER_DEFINED_NODE_BUFFER_DEPTH 1
```

enumerations:

```
typedef enum
  {
    BASIC,
    DATAGRAM,
    SNIP,
    STREAM

  } payload_type_enum_t;
```

Type definitions use a trailing *_t* to signify it is a type as as in the previous example *payload_type_enum_t.*

For loop and If statements and Functions:

All for loops and If statements will use full brackets regardless of the number of statements and have a blank space above and below each curly bracket

```
for (int i = 0; i < USER_DEFINED_CONSUMER_COUNT; i++) {

    openlcb_node->consumers.list[i] = 0;

  }

if (openlcb_nodes.count == 0) {

    return NULL;

} else {

}
```

Else If will use the following style

```
if (openlcb_nodes.count == 0) {

    return NULL;
```

```
} else if ( openlcb_nodes.count == 100)
   .. statements

} else {

   … statements

}
```

Functions have the first curly bracket on the same line as the function name and be on a separate line with on line of white space above and below both

```
void _generate_event_ids(openlcb_node_t* openlcb_node) {

}
```

Module functions:

within a module the following function and variable naming convention is used:

Any function/variable that is accessed outside the module it through the header file and using the name of the module in CaMeL case and lower case with underscores between works

```
can_msg_t* CanBufferStore_allocate_buffer(void);
```

Any function or GLOBAL variable within a module will lead with an underscore and be lower case separated by underscores

```
openlcb_nodes_t _openlcb_nodes;

void _generate_event_ids(openlcb_node_t* openlcb_node) {
}
```

variables in parameter lists or created in a function are lower case separated by underscores

```
void _generate_event_ids(openlcb_node_t* openlcb_node) {

   uint64_t node_id = openlcb_node->id << 16;
   uint16_t indexer = 0;

   …..

}
```