OpenLcbCLib

C Library to build OpenLcb/LCC Nodes on any hardware

Jim Kueneman 12/30/2025

Open Source C Based Software Stack

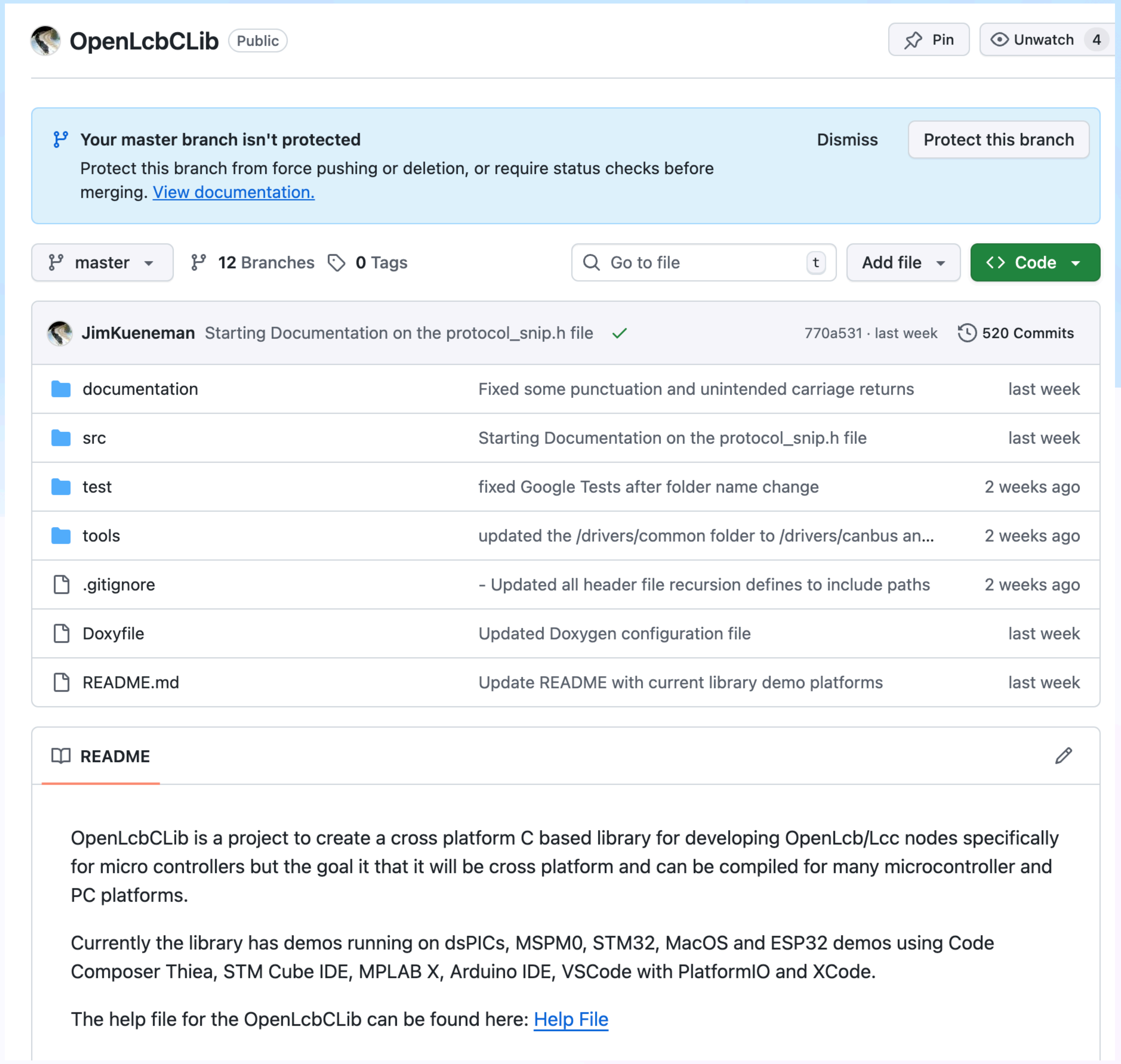
Abstraction for the OpenLCB/LCC Standards

- OpenLcb is the basis for the NMRA LCC (Layout Command Control) specifications
    - https://www.nmra.org/lcc
- Significant amount of code to write after understanding and implementing the specifications
- Software Stacks simplify application implementation
    - focus on coding your boards features *not* the OpenLcb/LCC specs
- OpenLcbCLib augments the OpenMRN C++ based code base for those who desire to use C and want to minimize flash usage
    - OpenMRN is powerful but takes resources
    - https://github.com/bakerstu/openmrn
- You can access the in-work help file here: https://jimkueneman.github.io/OpenLcbCLib/documentation/help/html/

GitHub Repository

Open Source

- https://github.com/JimKueneman/OpenLcbCLib
- documentation
  - design data files, style guides, tutorials, html help generated from Doxygen
- src
  - demos, library source code, templates for applications, Google test infrastructure
- test
  - Goole test infrastructure
- tools
  - tools to help develop applications



Your master branch isn't protected

Protect this branch from force pushing or deletion, or require status checks before merging. View documentation.

JimKueneman Starting Documentation on the protocol_snip.h file ✓ 770a531 · last week 520 Commits

documentation	Fixed some punctuation and unintended carriage returns	last week
src	Starting Documentation on the protocol_snip.h file	last week
test	fixed Google Tests after folder name change	2 weeks ago
tools	updated the /drivers/common folder to /drivers/canbus an...	2 weeks ago
.gitignore	- Updated all header file recursion defines to include paths	2 weeks ago
Doxyfile	Updated Doxygen configuration file	last week
README.md	Update README with current library demo platforms	last week

OpenLcbCLib is a project to create a cross platform C based library for developing OpenLcb/Lcc nodes specifically for micro controllers but the goal it that it will be cross platform and can be compiled for many microcontroller and PC platforms.

Currently the library has demos running on dsPICs, MSPM0, STM32, MacOS and ESP32 demos using Code Composer Thiea, STM Cube IDE, MPLAB X, Arduino IDE, VSCode with PlatformIO and XCode.

The help file for the OpenLcbCLib can be found here: Help File

Core Library File Structure Overview

Simple straightforward file structure

src/ openlcb

- Core OpenLcb message state machines
- Core OpenLcb message handlers
- OpenLcb message data buffers and

src/drivers

- Root of various drivers that are used on the physical layer (i.e. CAN, TCP/IP, Bluetooth, etc)

src/drivers/ CANBUS

- Core CAN message state machines
- Core CAN message handlers
- CAN message data buffers and structures

src/ applications

- Examples of a Basic Node that implement all common OpenLcb protocols on various platforms and devices (Arduino, PlatformIO, TI Code Composer Theia,

src/test

- Core Google Test folder for building and executing built in tests for the modules, 95%+ coverage

Task/State Machine/Handler Architecture

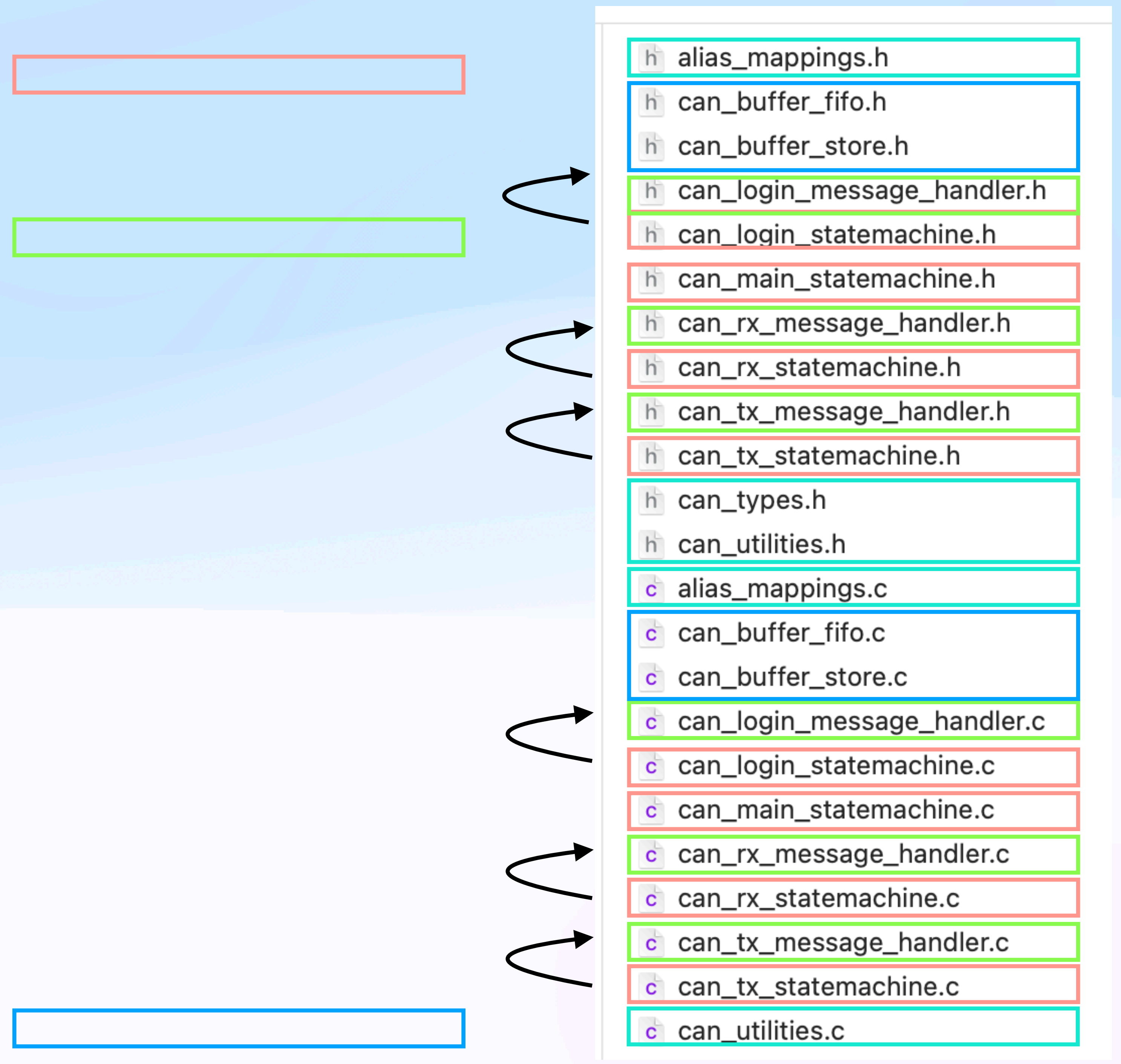
Core Tasks

- There are several main tasks that need to occur in an OpenLcb/LCC application, we will discuss the CAN bus implementation:
    - CAN Buffer Pool and Lists/FIFOs
    - CAN Frame Receive (Rx)
    - CAN Frame Transmit(Tx)
    - CAN Login (Alias allocation)
    - CAN Message Loop
- In the OpenLcb/LCC implementation:
    - OpenLcb/LCC Buffer Pool and Lists/FIFOs
    - OpenLcb/LCC Login (Initialization complete/Consumers/Producer announcement)
    - OpenLcb/LCC Message Loop (dispatching incoming OpenLcb/LCC messages to the Node(s))
    - OpenLcb/LCC Node Pool and List

State machine/Handler Architecture

CAN Driver Implementation

- Each major task has a state machine module
- Each state machine has one or more associated handler modules
  - handlers are called to execute the task requested by the state machine
  - state machine modules are switch statements that call the handler functions
- CAN message buffers and lists
- Types/Structures/Helper Utilities for CAN messages and structures

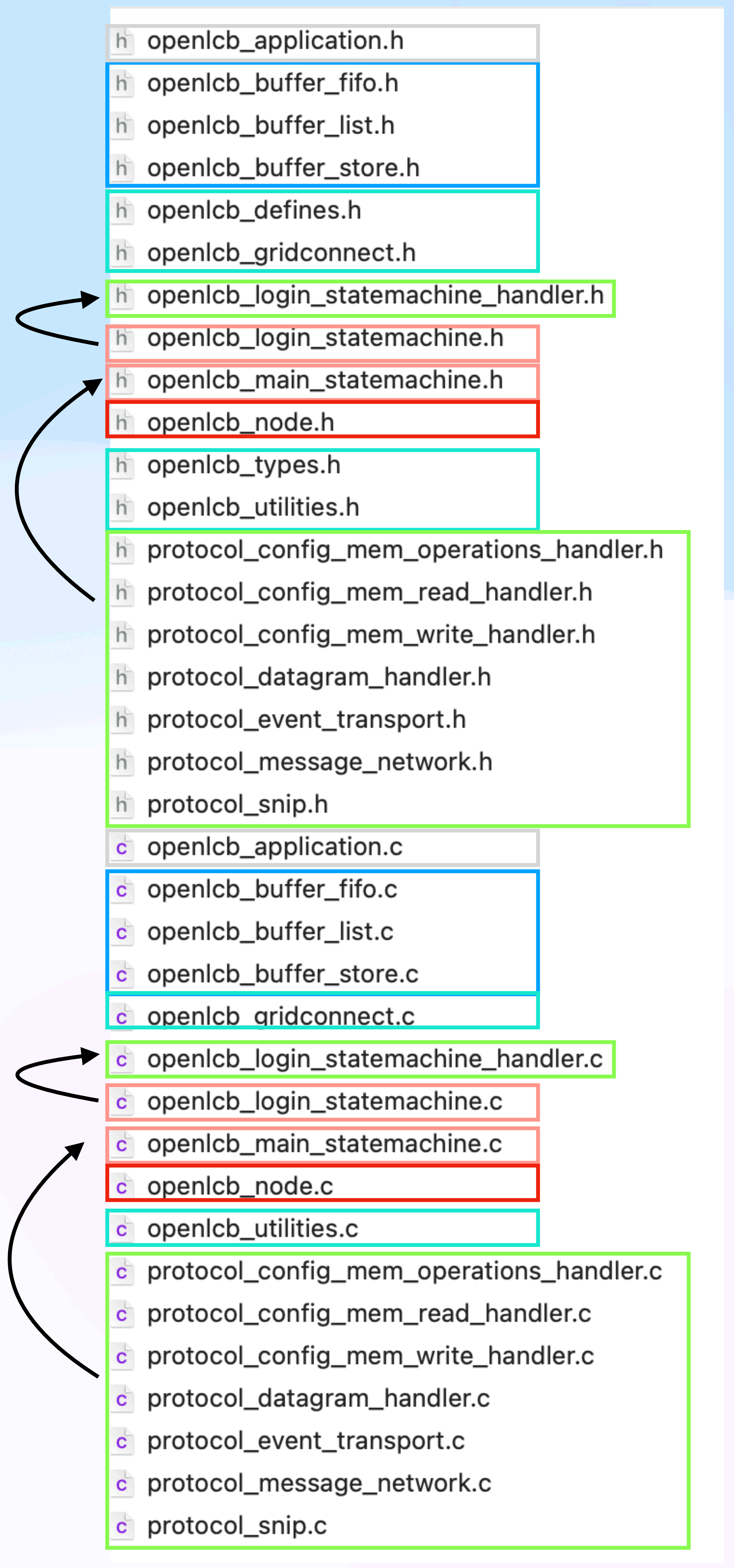


```
alias_mappings.h
can_buffer_fifo.h
can_buffer_store.h
can_login_message_handler.h
can_login_statemachine.h
can_main_statemachine.h
can_rx_message_handler.h
can_rx_statemachine.h
can_tx_message_handler.h
can_tx_statemachine.h
can_types.h
can_utilities.h
alias_mappings.c
can_buffer_fifo.c
can_buffer_store.c
can_login_message_handler.c
can_login_statemachine.c
can_main_statemachine.c
can_rx_message_handler.c
can_rx_statemachine.c
can_tx_message_handler.c
can_tx_statemachine.c
can_utilities.c
```

State machine/Handler Architecture

OpenLcb/LCC Implementation

- Each major task has a state machine module 🟧
- Each state machine has one or more associated handler modules 🟩
    - handlers are called to execute the task requested by the state machine
    - state machine modules are switch statements that call the handler functions
- OpenLcb/LCC message buffers and lists 🟦
- Types/Structures/Helper Utilities for CAN messages and structures 🟦
- OpenLcb/LCC Node structures 🟥
- Application Level Helper Functions (main interface for an application) ⬜



```
h openlcb_application.h
h openlcb_buffer_fifo.h
h openlcb_buffer_list.h
h openlcb_buffer_store.h
h openlcb_defines.h
h openlcb_gridconnect.h
h openlcb_login_statemachine_handler.h
h openlcb_login_statemachine.h
h openlcb_main_statemachine.h
h openlcb_node.h
h openlcb_types.h
h openlcb_utilities.h
h protocol_config_mem_operations_handler.h
h protocol_config_mem_read_handler.h
h protocol_config_mem_write_handler.h
h protocol_datagram_handler.h
h protocol_event_transport.h
h protocol_message_network.h
h protocol_snip.h
c openlcb_application.c
c openlcb_buffer_fifo.c
c openlcb_buffer_list.c
c openlcb_buffer_store.c
c openlcb_gridconnect.c
c openlcb_login_statemachine_handler.c
c openlcb_login_statemachine.c
c openlcb_main_statemachine.c
c openlcb_node.c
c openlcb_utilities.c
c protocol_config_mem_operations_handler.c
c protocol_config_mem_read_handler.c
c protocol_config_mem_write_handler.c
c protocol_datagram_handler.c
c protocol_event_transport.c
c protocol_message_network.c
c protocol_snip.c
```

Dependency Injection

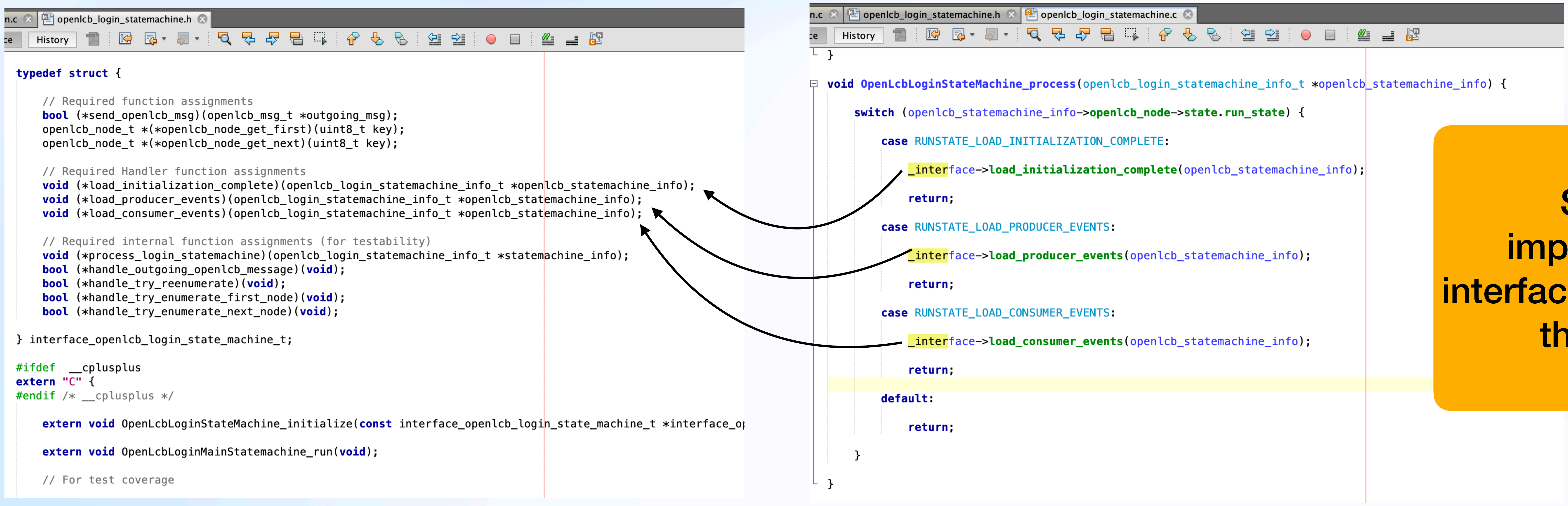
Allows for Complete Testing and Low Level Application Hooking

- The Library utilizes function pointers interfaces for each module/unit:
- Assigned and resolved by the application during compile time
- Eliminates coupling between modules/units (#includes not required to access other modules/units)
- Allows the Google Test to implement external functions that can inject correct/incorrect results to completely test a module/unit without depending on other modules/units
- Allow setting the function pointer to NULL to not implement a feature and strip the code out of the final executable minimizing image size

Interface Function Pointer Structures

External Functions Required by a Module/Unit are called through the Interface

- At the top of each module/unit header file is an interface struct defining the functions that the module/unit will require to perform an outside action
  - Note: Some modules create an entry in this structure that is assigned a function within the same module, this allows complete test coverage by injecting test only implementations of these function



```c
typedef struct {

    // Required function assignments
    bool (*send_openlcb_msg)(openlcb_msg_t *outgoing_msg);
    openlcb_node_t *(*openlcb_node_get_first)(uint8_t key);
    openlcb_node_t *(*openlcb_node_get_next)(uint8_t key);

    // Required Handler function assignments
    void (*load_initialization_complete)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);
    void (*load_producer_events)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);
    void (*load_consumer_events)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);

    // Required internal function assignments (for testability)
    void (*process_login_statemachine)(openlcb_login_statemachine_info_t *statemachine_info);
    bool (*handle_outgoing_openlcb_message)(void);
    bool (*handle_try_reenumerate)(void);
    bool (*handle_try_enumerate_first_node)(void);
    bool (*handle_try_enumerate_next_node)(void);

} interface_openlcb_login_state_machine_t;

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

    extern void OpenLcbLoginStateMachine_initialize(const interface_openlcb_login_state_machine_t *interface_o

    extern void OpenLcbLoginMainStatemachine_run(void);

    // For test coverage
```

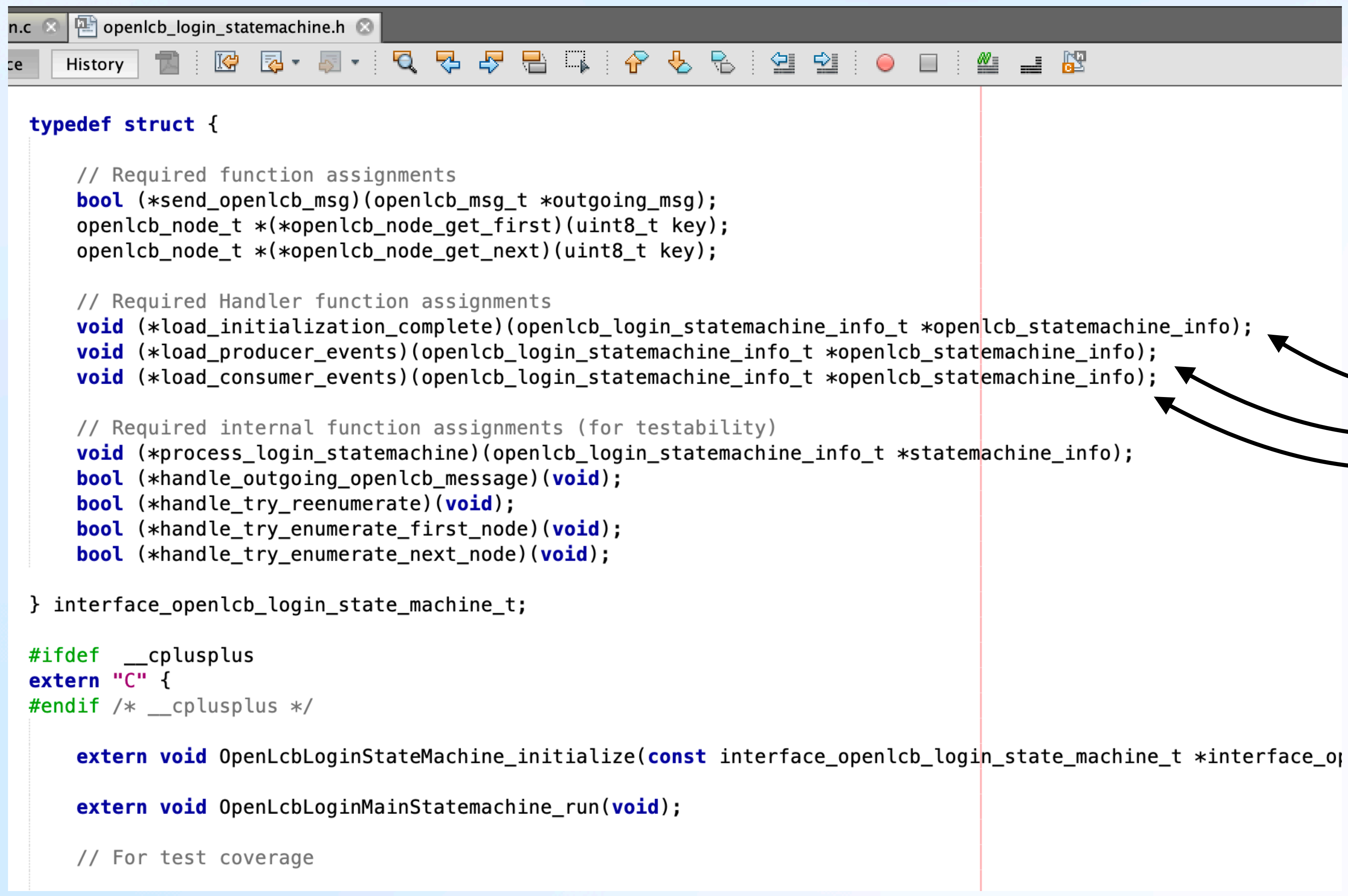
```c
}

void OpenLcbLoginStateMachine_process(openlcb_login_statemachine_info_t *openlcb_statemachine_info) {

    switch (openlcb_statemachine_info->openlcb_node->state.run_state) {

        case RUNSTATE_LOAD_INITIALIZATION_COMPLETE:

            _interface->load_initialization_complete(openlcb_statemachine_info);

            return;

        case RUNSTATE_LOAD_PRODUCER_EVENTS:

            _interface->load_producer_events(openlcb_statemachine_info);

            return;

        case RUNSTATE_LOAD_CONSUMER_EVENTS:

            _interface->load_consumer_events(openlcb_statemachine_info);

            return;

        default:

            return;

    }

}
```

State machine implementations use interface functions to handle the task required

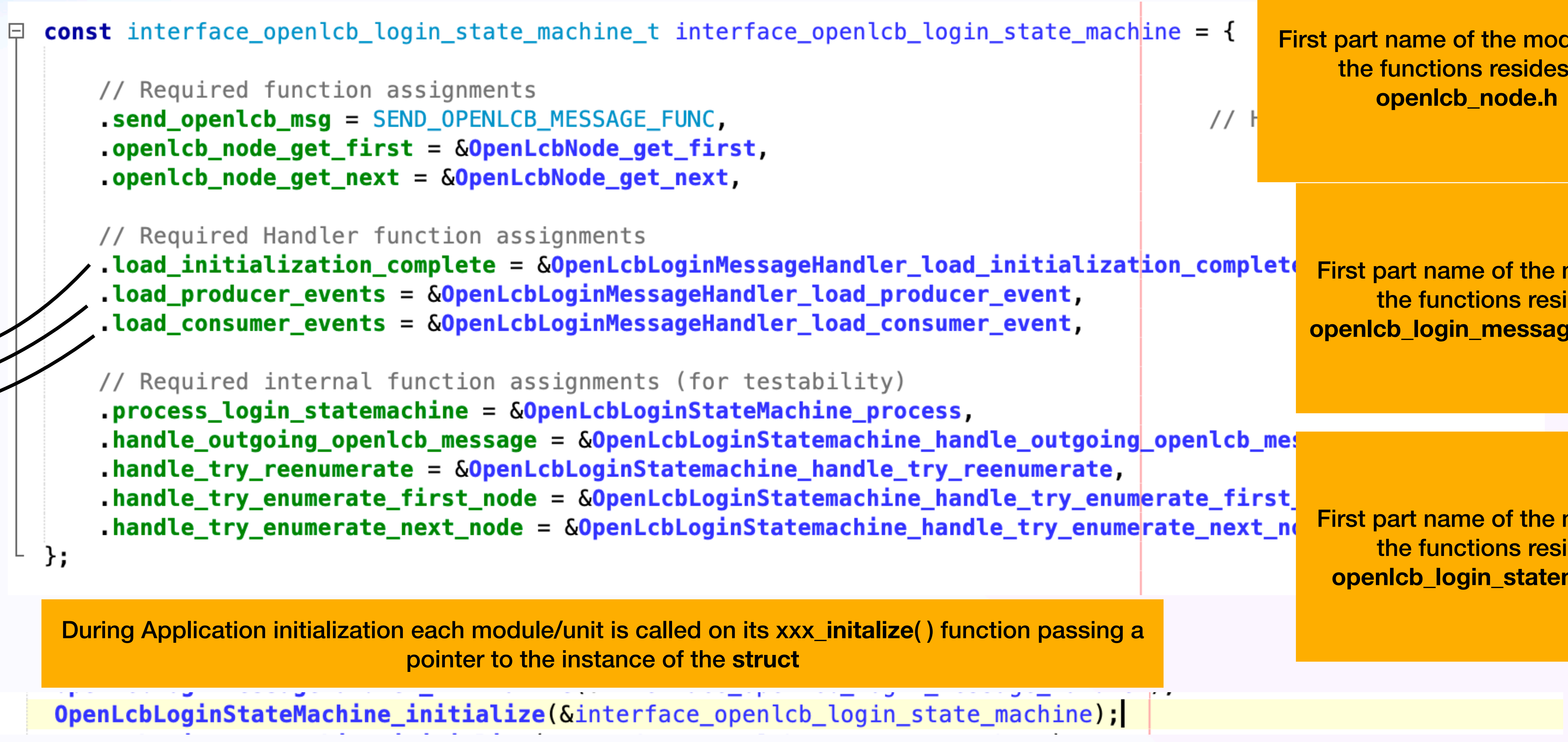
Assigning Interface Function Pointers

Most Assignments are to Library Functions in Other Modules/Units

- The Application creates an instance of each interface and assigns the dependent function pointer to the fields
  - Defining them as **const** allow these jump table to be stored in FLASH saving precious RAM in microcontrollers.



```c
typedef struct {

    // Required function assignments
    bool (*send_openlcb_msg)(openlcb_msg_t *outgoing_msg);
    openlcb_node_t *(*openlcb_node_get_first)(uint8_t key);
    openlcb_node_t *(*openlcb_node_get_next)(uint8_t key);

    // Required Handler function assignments
    void (*load_initialization_complete)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);
    void (*load_producer_events)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);
    void (*load_consumer_events)(openlcb_login_statemachine_info_t *openlcb_statemachine_info);

    // Required internal function assignments (for testability)
    void (*process_login_statemachine)(openlcb_login_statemachine_info_t *statemachine_info);
    bool (*handle_outgoing_openlcb_message)(void);
    bool (*handle_try_reenumerate)(void);
    bool (*handle_try_enumerate_first_node)(void);
    bool (*handle_try_enumerate_next_node)(void);

} interface_openlcb_login_state_machine_t;

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

    extern void OpenLcbLoginStateMachine_initialize(const interface_openlcb_login_state_machine_t *interface_op

    extern void OpenLcbLoginMainStatemachine_run(void);

    // For test coverage
```



```c
const interface_openlcb_login_state_machine_t interface_openlcb_login_state_machine = {

    // Required function assignments
    .send_openlcb_msg = SEND_OPENLCB_MESSAGE_FUNC,                                // H
    .openlcb_node_get_first = &OpenLcbNode_get_first,
    .openlcb_node_get_next = &OpenLcbNode_get_next,

    // Required Handler function assignments
    .load_initialization_complete = &OpenLcbLoginMessageHandler_load_initialization_complete
    .load_producer_events = &OpenLcbLoginMessageHandler_load_producer_event,
    .load_consumer_events = &OpenLcbLoginMessageHandler_load_consumer_event,

    // Required internal function assignments (for testability)
    .process_login_statemachine = &OpenLcbLoginStateMachine_process,
    .handle_outgoing_openlcb_message = &OpenLcbLoginStatemachine_handle_outgoing_openlcb_mes
    .handle_try_reenumerate = &OpenLcbLoginStatemachine_handle_try_reenumerate,
    .handle_try_enumerate_first_node = &OpenLcbLoginStatemachine_handle_try_enumerate_first_
    .handle_try_enumerate_next_node = &OpenLcbLoginStatemachine_handle_try_enumerate_next_n
};
```

```c
OpenLcbLoginStateMachine_initialize(&interface_openlcb_login_state_machine);
```

First part name of the module/unit the functions resides in **openlcb_node.h**

First part name of the module/unit the functions resides in **openlcb_login_message_handler.h**

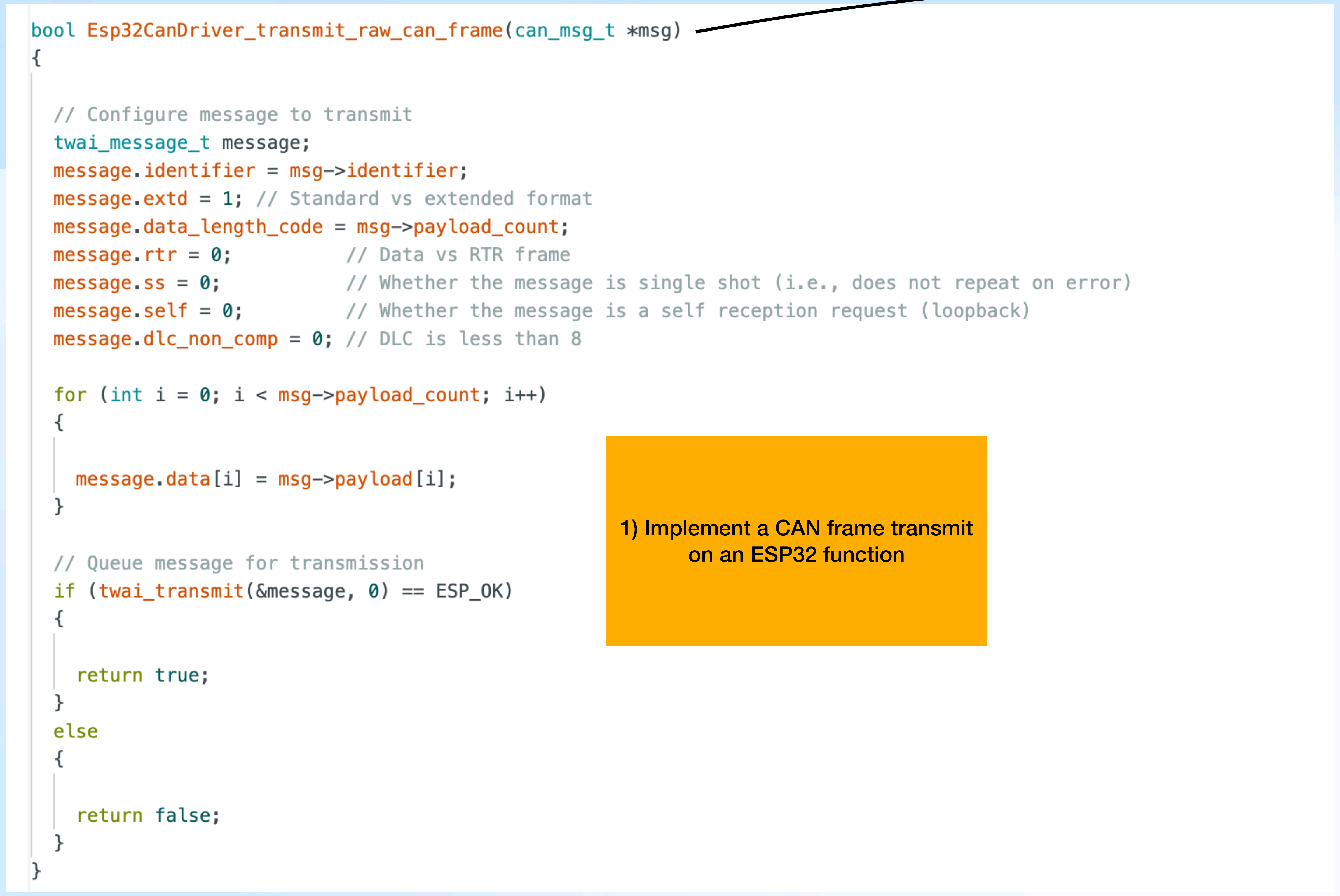
First part name of the module/unit the functions resides in **openlcb_login_statemachine.h**

During Application initialization each module/unit is called on its xxx_**initalize**( ) function passing a pointer to the instance of the **struct**

Application Level Drivers

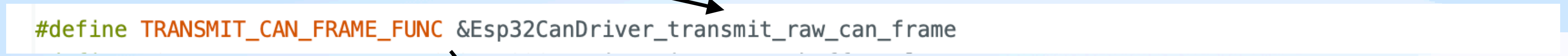
Same pattern is used for Application Drivers

- Drivers implement required task in a function and that function is assigned to the appropriate interface function pointer



```
bool Esp32CanDriver_transmit_raw_can_frame(can_msg_t *msg)
{
  // Configure message to transmit
  twai_message_t message;
  message.identifier = msg->identifier;
  message.extd = 1; // Standard vs extended format
  message.data_length_code = msg->payload_count;
  message.rtr = 0;          // Data vs RTR frame
  message.ss = 0;           // Whether the message is single shot (i.e., does not repeat on error)
  message.self = 0;         // Whether the message is a self reception request (loopback)
  message.dlc_non_comp = 0; // DLC is less than 8

  for (int i = 0; i < msg->payload_count; i++)
  {
    message.data[i] = msg->payload[i];
  }

  // Queue message for transmission
  if (twai_transmit(&message, 0) == ESP_OK)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

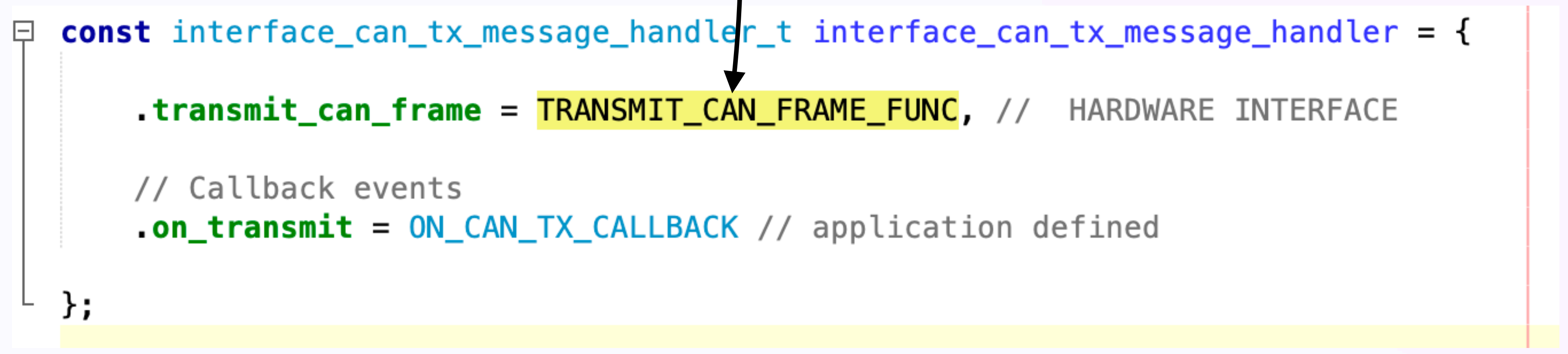
1) Implement a CAN frame transmit on an ESP32 function



```
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
```

2) Assign the function to the define that the library requires to send a frame to the CAN wire

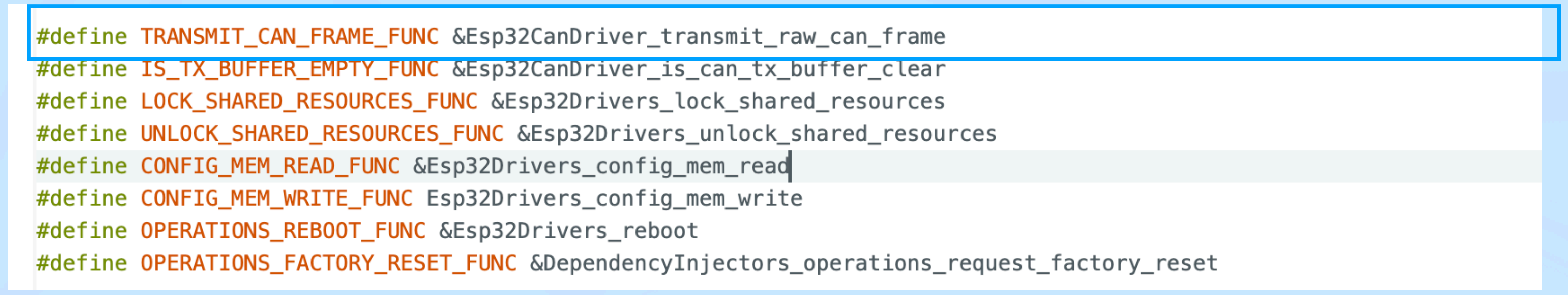
Now the dependency for the can_tx_message_handler.h is linked to the Application's hardware to transmit a CAN frame



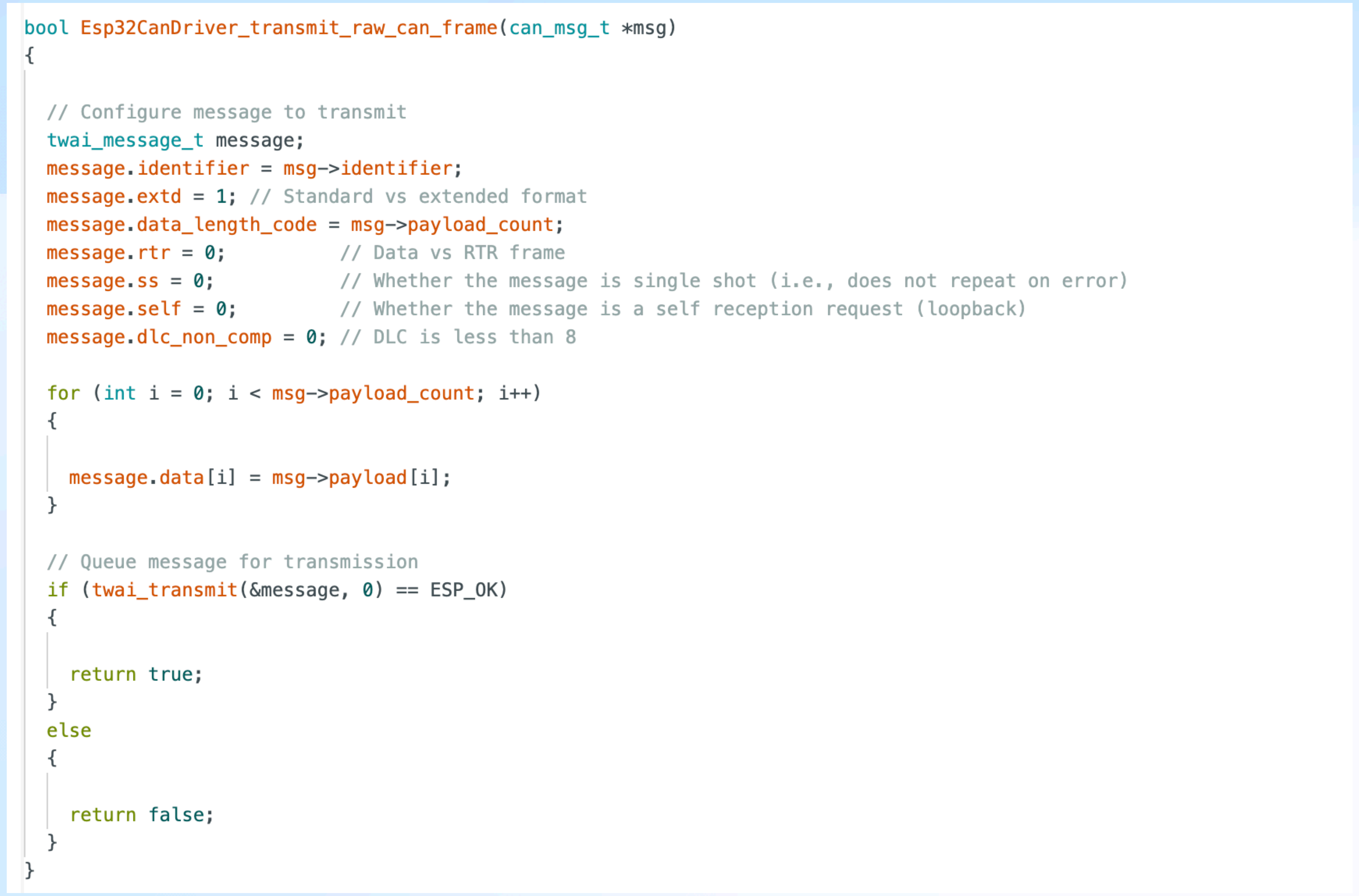
```
const interface_can_tx_message_handler_t interface_can_tx_message_handler = {

    .transmit_can_frame = TRANSMIT_CAN_FRAME_FUNC, //  HARDWARE INTERFACE

    // Callback events
    .on_transmit = ON_CAN_TX_CALLBACK // application defined
};
```

Eight Application Defined Functions Required

Driver - Transmit CAN Frame



```
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
#define IS_TX_BUFFER_EMPTY_FUNC &Esp32CanDriver_is_can_tx_buffer_clear
#define LOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_lock_shared_resources
#define UNLOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_unlock_shared_resources
#define CONFIG_MEM_READ_FUNC &Esp32Drivers_config_mem_read
#define CONFIG_MEM_WRITE_FUNC Esp32Drivers_config_mem_write
#define OPERATIONS_REBOOT_FUNC &Esp32Drivers_reboot
#define OPERATIONS_FACTORY_RESET_FUNC &DependencyInjectors_operations_request_factory_reset
```

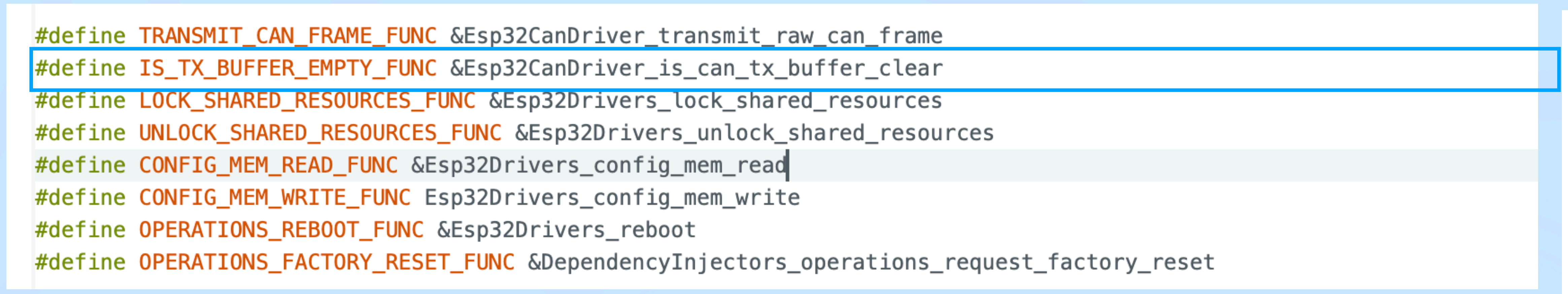


```
bool Esp32CanDriver_transmit_raw_can_frame(can_msg_t *msg)
{

  // Configure message to transmit
  twai_message_t message;
  message.identifier = msg->identifier;
  message.extd = 1; // Standard vs extended format
  message.data_length_code = msg->payload_count;
  message.rtr = 0;          // Data vs RTR frame
  message.ss = 0;           // Whether the message is single shot (i.e., does not repeat on error)
  message.self = 0;         // Whether the message is a self reception request (loopback)
  message.dlc_non_comp = 0; // DLC is less than 8

  for (int i = 0; i < msg->payload_count; i++)
  {

    message.data[i] = msg->payload[i];
  }

  // Queue message for transmission
  if (twai_transmit(&message, 0) == ESP_OK)
  {

    return true;
  }
  else
  {

    return false;
  }
}
```

- Passes a CAN message that needs to be sent
- Return True if the message is buffered to send, False if it fails

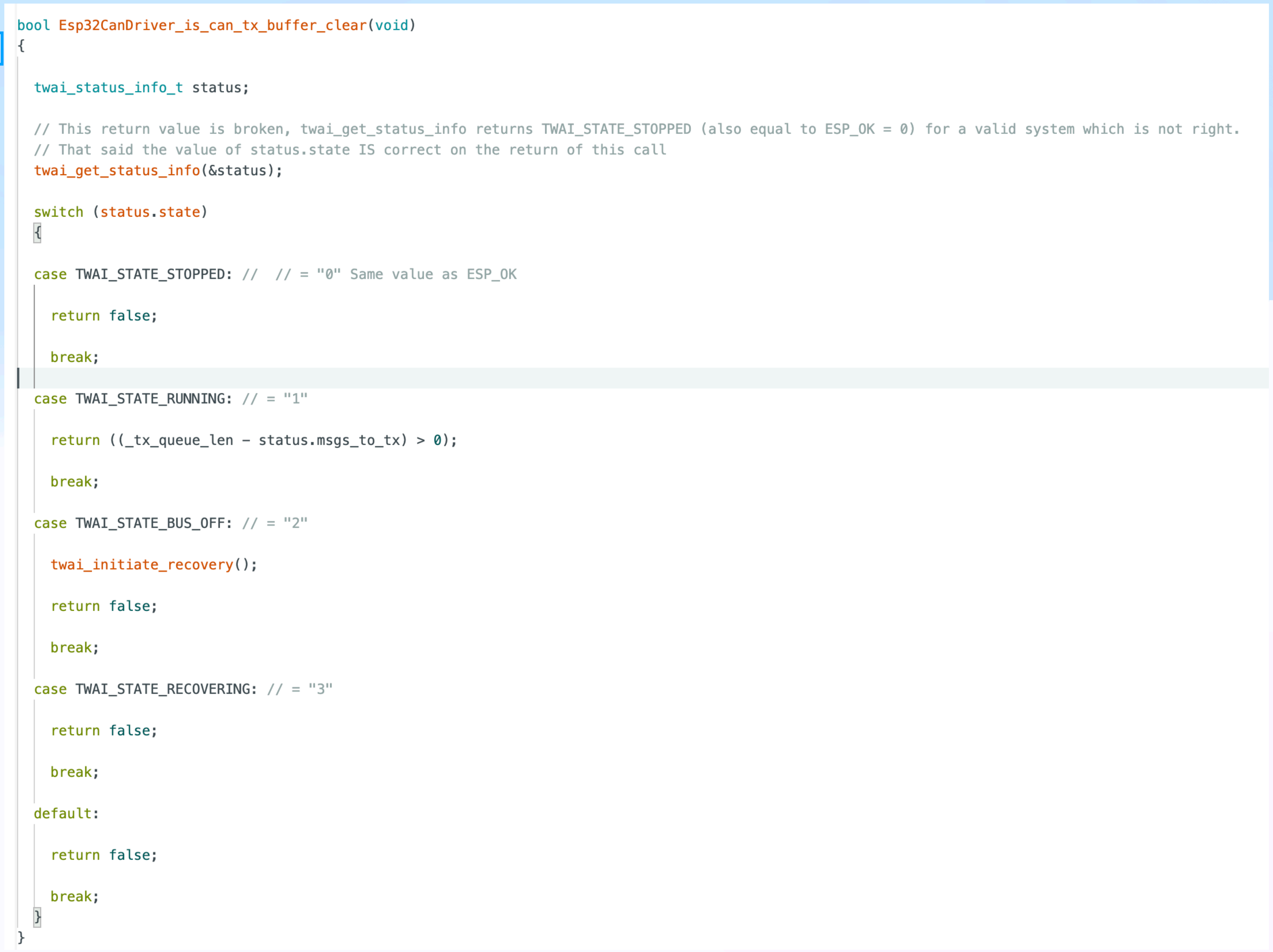
Eight Application Defined Functions Required

Driver - Is Transmit CAN Buffer Empty



```c
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
#define IS_TX_BUFFER_EMPTY_FUNC &Esp32CanDriver_is_can_tx_buffer_clear
#define LOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_lock_shared_resources
#define UNLOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_unlock_shared_resources
#define CONFIG_MEM_READ_FUNC &Esp32Drivers_config_mem_read
#define CONFIG_MEM_WRITE_FUNC Esp32Drivers_config_mem_write
#define OPERATIONS_REBOOT_FUNC &Esp32Drivers_reboot
#define OPERATIONS_FACTORY_RESET_FUNC &DependencyInjectors_operations_request_factory_reset
```

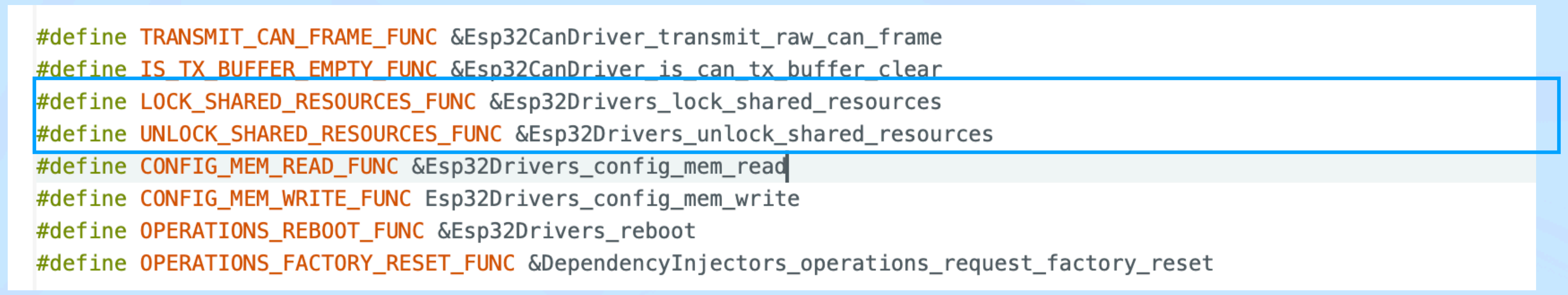
- Return True if the hardware transmit buffer is empty and a call to TRANSMIT_CAN_FRAME_FUNC will succeed, False if not



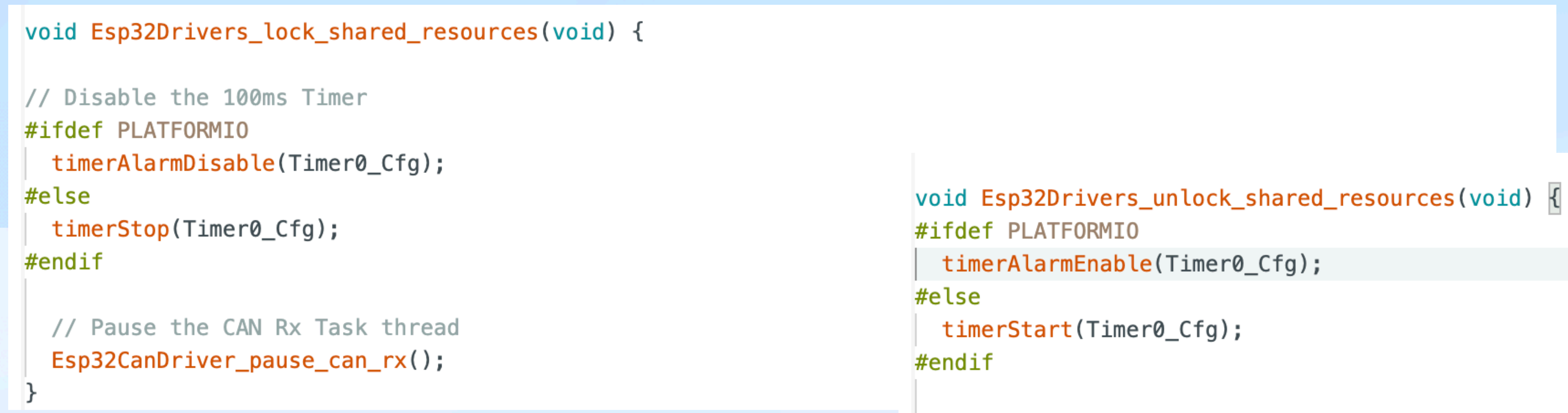
```c
bool Esp32CanDriver_is_can_tx_buffer_clear(void)
{

  twai_status_info_t status;

  // This return value is broken, twai_get_status_info returns TWAI_STATE_STOPPED (also equal to ESP_OK = 0) for a valid system which is not right.
  // That said the value of status.state IS correct on the return of this call
  twai_get_status_info(&status);

  switch (status.state)
  {

  case TWAI_STATE_STOPPED: //  // = "0" Same value as ESP_OK

    return false;

    break;

  case TWAI_STATE_RUNNING: // = "1"

    return ((_tx_queue_len - status.msgs_to_tx) > 0);

    break;

  case TWAI_STATE_BUS_OFF: // = "2"

    twai_initiate_recovery();

    return false;

    break;

  case TWAI_STATE_RECOVERING: // = "3"

    return false;

    break;

  default:

    return false;

    break;
  }
}
```

Eight Application Defined Functions Required

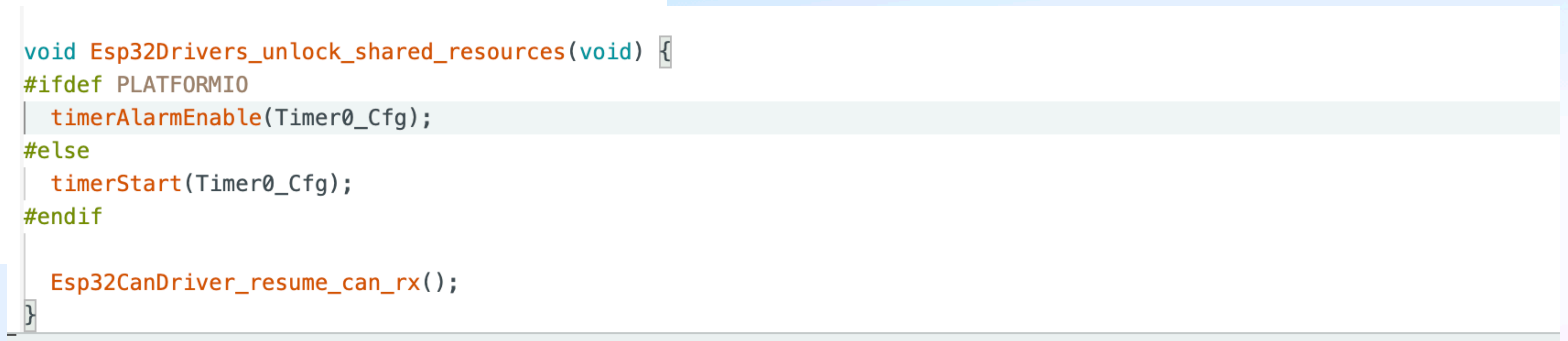
Driver - Lock Shared Resources



```
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
#define IS_TX_BUFFER_EMPTY_FUNC &Esp32CanDriver_is_can_tx_buffer_clear
#define LOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_lock_shared_resources
#define UNLOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_unlock_shared_resources
#define CONFIG_MEM_READ_FUNC &Esp32Drivers_config_mem_read
#define CONFIG_MEM_WRITE_FUNC Esp32Drivers_config_mem_write
#define OPERATIONS_REBOOT_FUNC &Esp32Drivers_reboot
#define OPERATIONS_FACTORY_RESET_FUNC &DependencyInjectors_operations_request_factory_reset
```



```
void Esp32Drivers_lock_shared_resources(void) {

// Disable the 100ms Timer
#ifdef PLATFORMIO
  timerAlarmDisable(Timer0_Cfg);
#else
  timerStop(Timer0_Cfg);
#endif

  // Pause the CAN Rx Task thread
  Esp32CanDriver_pause_can_rx();
}
```

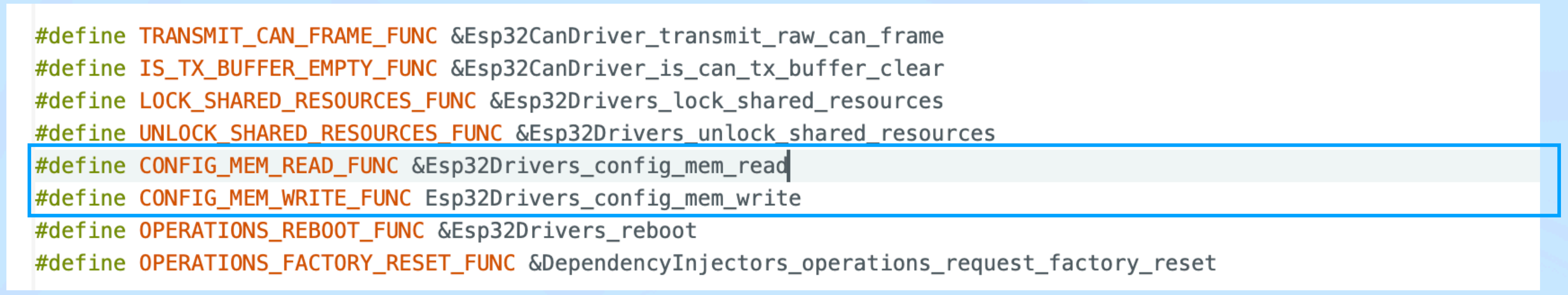


```
void Esp32Drivers_unlock_shared_resources(void) {
#ifdef PLATFORMIO
  timerAlarmEnable(Timer0_Cfg);
#else
  timerStart(Timer0_Cfg);
#endif

  Esp32CanDriver_resume_can_rx();
}
```

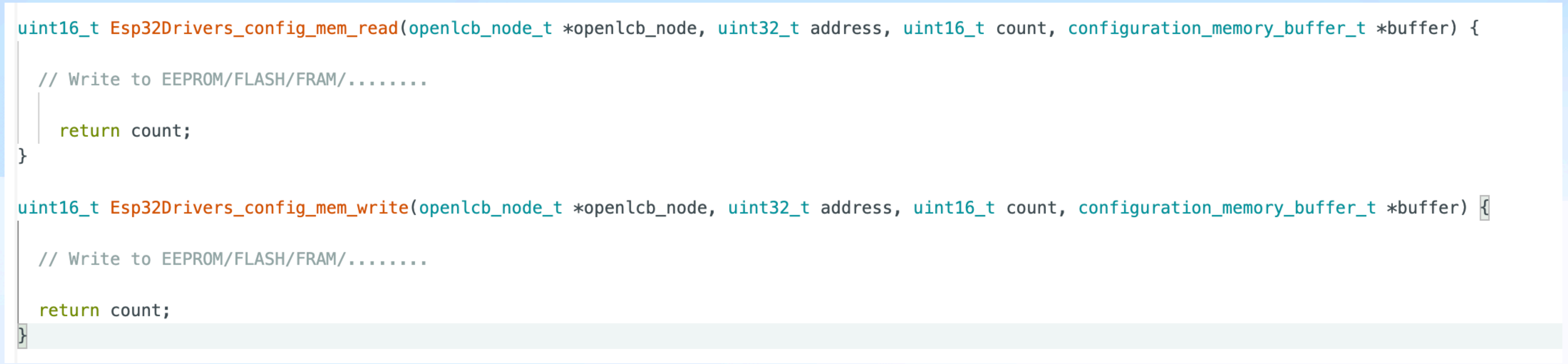
- The assumption is that messages are received in an interrupt or thread
- The receive state machine must allocate and manipulate buffers that the main loop also must access so race conditions are possible
- The 100ms Timer is also assumed to be in an interrupt or thread and can end up with race conditions
- Neither the message receive or 100ms timer tick negotiate for exclusive buffer access and assume they are always safe to access those resources
- A call to this function must stop the message receive interrupt and 100ms timer from calling into the library until the UnLock Resources call is seen
- The time between Lock and Unlock is _very_ short. Typically less than 10 clock cycles so nothing fancy is required. For the ESP32 drive code above the Timer is stopped and the Task Thread running the receive code is paused. In the XCode implementation with threads just sets a flag and and does not call into the library from these threads until an UnLock is received.

Eight Application Defined Functions Required

Driver - Configuration Memory Read/Write



```
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
#define IS_TX_BUFFER_EMPTY_FUNC &Esp32CanDriver_is_can_tx_buffer_clear
#define LOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_lock_shared_resources
#define UNLOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_unlock_shared_resources
#define CONFIG_MEM_READ_FUNC &Esp32Drivers_config_mem_read
#define CONFIG_MEM_WRITE_FUNC Esp32Drivers_config_mem_write
#define OPERATIONS_REBOOT_FUNC &Esp32Drivers_reboot
#define OPERATIONS_FACTORY_RESET_FUNC &DependencyInjectors_operations_request_factory_reset
```

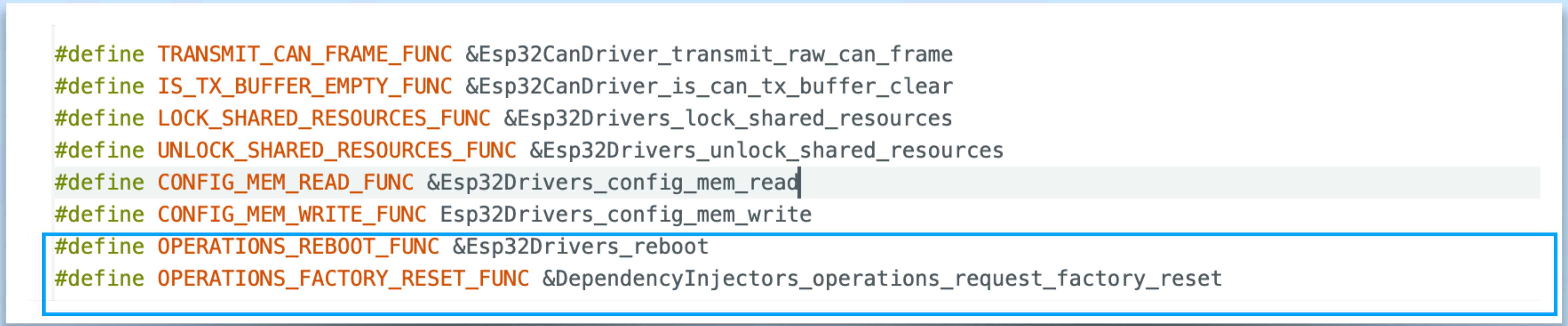


```
uint16_t Esp32Drivers_config_mem_read(openlcb_node_t *openlcb_node, uint32_t address, uint16_t count, configuration_memory_buffer_t *buffer) {

  // Write to EEPROM/FLASH/FRAM/........

    return count;
}

uint16_t Esp32Drivers_config_mem_write(openlcb_node_t *openlcb_node, uint32_t address, uint16_t count, configuration_memory_buffer_t *buffer) {

  // Write to EEPROM/FLASH/FRAM/........

  return count;
}
```

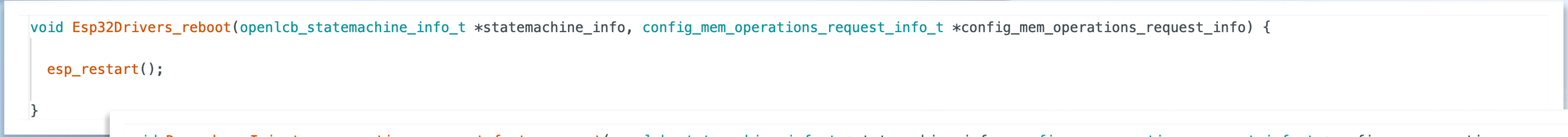
- openlcb_node: Pointer to the Node that is being requested to access to its configuration memory
- address: Configuration Memory address requested to read/write
- count: Number of bytes to read/write
- buffer: Pointer to the byte array to either read from or write to

Eight Application Defined Functions Required

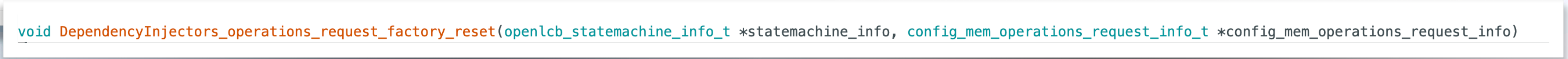
Driver - Reboot



```
#define TRANSMIT_CAN_FRAME_FUNC &Esp32CanDriver_transmit_raw_can_frame
#define IS_TX_BUFFER_EMPTY_FUNC &Esp32CanDriver_is_can_tx_buffer_clear
#define LOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_lock_shared_resources
#define UNLOCK_SHARED_RESOURCES_FUNC &Esp32Drivers_unlock_shared_resources
#define CONFIG_MEM_READ_FUNC &Esp32Drivers_config_mem_read
#define CONFIG_MEM_WRITE_FUNC Esp32Drivers_config_mem_write
#define OPERATIONS_REBOOT_FUNC &Esp32Drivers_reboot
#define OPERATIONS_FACTORY_RESET_FUNC &DependencyInjectors_operations_request_factory_reset
```



```
void Esp32Drivers_reboot(openlcb_statemachine_info_t *statemachine_info, config_mem_operations_request_info_t *config_mem_operations_request_info) {

  esp_restart();

}
```

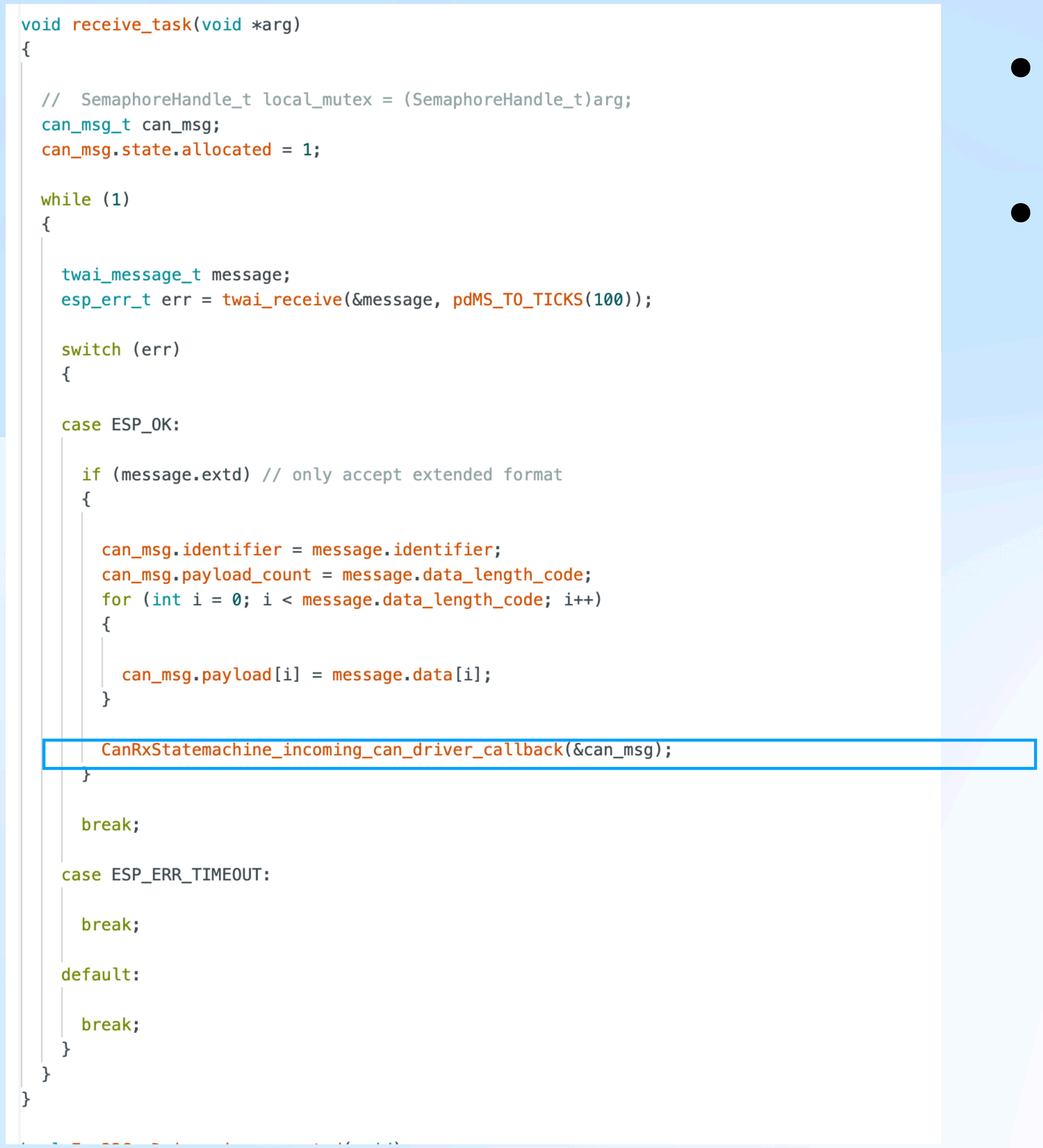


```
void DependencyInjectors_operations_request_factory_reset(openlcb_statemachine_info_t *statemachine_info, config_mem_operations_request_info_t *config_mem_operations_request_info)
```

- Called when the Configuration Memory Protocol is called to Reboot the Node or to Reset the configuration to factory defaults
- statemachine_info: Pointer to a structure that carries a pointer to the Node being requested and the raw OpenLcb/LCC message buffer. The outgoing message in the structure is not used in this call
- config_mem_operations_request_info: Pointer to a structure that carries the details about the message. Most of the information is contained within the incoming message but the library pulls the details into this structure so it is easier to use and decode

Two Library Calls Required

Call - CAN Frame Rx

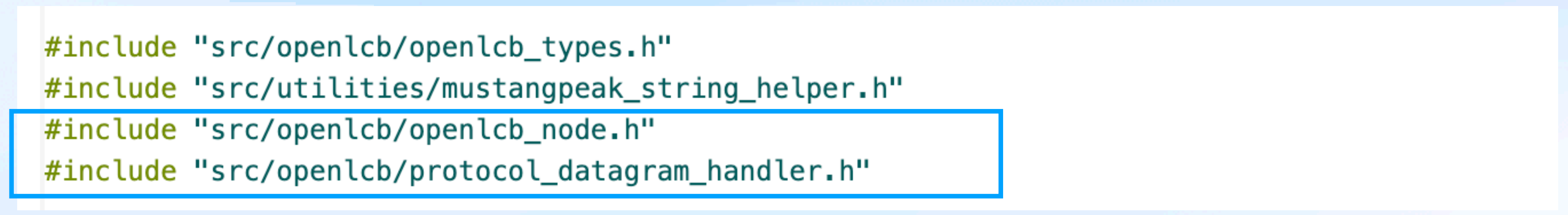


```
void receive_task(void *arg)
{

  //  SemaphoreHandle_t local_mutex = (SemaphoreHandle_t)arg;
  can_msg_t can_msg;
  can_msg.state.allocated = 1;

  while (1)
  {

    twai_message_t message;
    esp_err_t err = twai_receive(&message, pdMS_TO_TICKS(100));

    switch (err)
    {

    case ESP_OK:

      if (message.extd) // only accept extended format
      {

        can_msg.identifier = message.identifier;
        can_msg.payload_count = message.data_length_code;
        for (int i = 0; i < message.data_length_code; i++)
        {

          can_msg.payload[i] = message.data[i];
        }

        CanRxStatemachine_incoming_can_driver_callback(&can_msg);
      }

      break;

    case ESP_ERR_TIMEOUT:

      break;

    default:

      break;
    }
  }
}
```

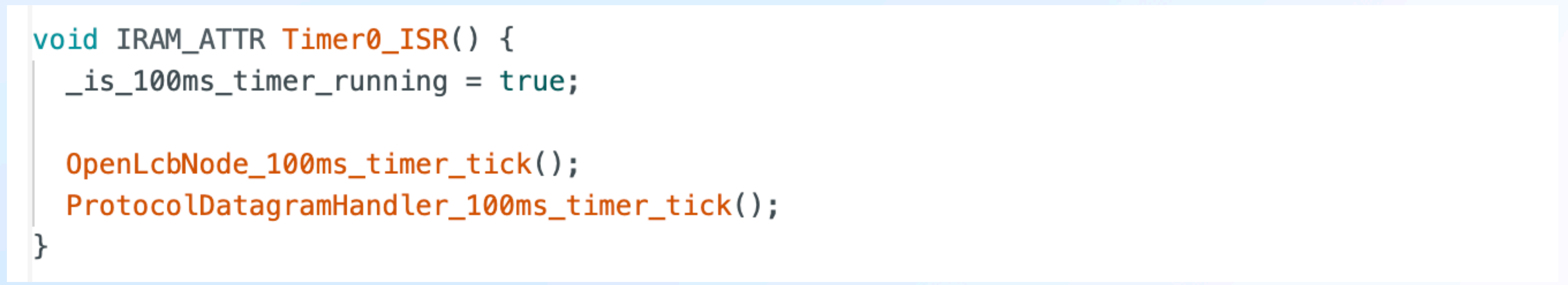
- Application defined receive function for a CAN frame
- Once successfully received the information is moved into a local can_msg_t structure and a callback into the library is used:
  - CanRxStatemachine_incoming_can_driver_callback(&can_msg)

Two Library Calls Required

Call - 100ms Timer Tick



```
#include "src/openlcb/openlcb_types.h"
#include "src/utilities/mustangpeak_string_helper.h"
#include "src/openlcb/openlcb_node.h"
#include "src/openlcb/protocol_datagram_handler.h"
```



```
void IRAM_ATTR Timer0_ISR() {
  _is_100ms_timer_running = true;

  OpenLcbNode_100ms_timer_tick();
  ProtocolDatagramHandler_100ms_timer_tick();
}
```

- Call two library functions with each timer tick
- OpenLcbNode_100ms_timer_tick( ) updates the time tick of each Node instance
  - used for various timing tasks including the wait during a login to a CAN network to allocate the Alias ID
- ProtocolDatagramHandler_100ms_timer_tick( ) looks through allocated messages for buffer allocations that have been allocated an extended time
  - Indicates that something happened to the source Node sending the message (throttle unplugged for instance) and the buffer gathering the multiple CAN frames is now abandoned and needs to be released.