UDACITY

Nanodegree

Automotive door control design

Ahmed Elebaby

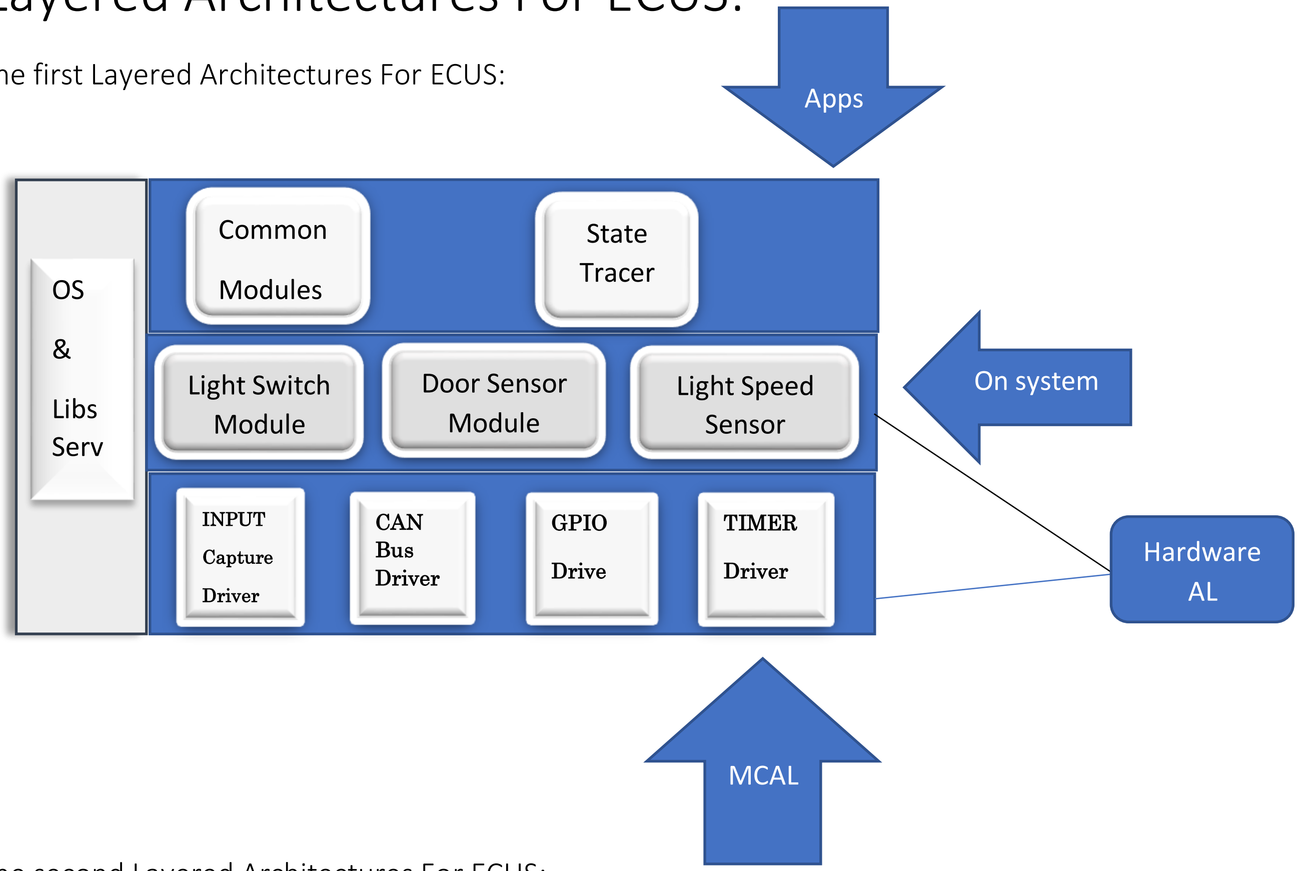
Static design

Introduction:

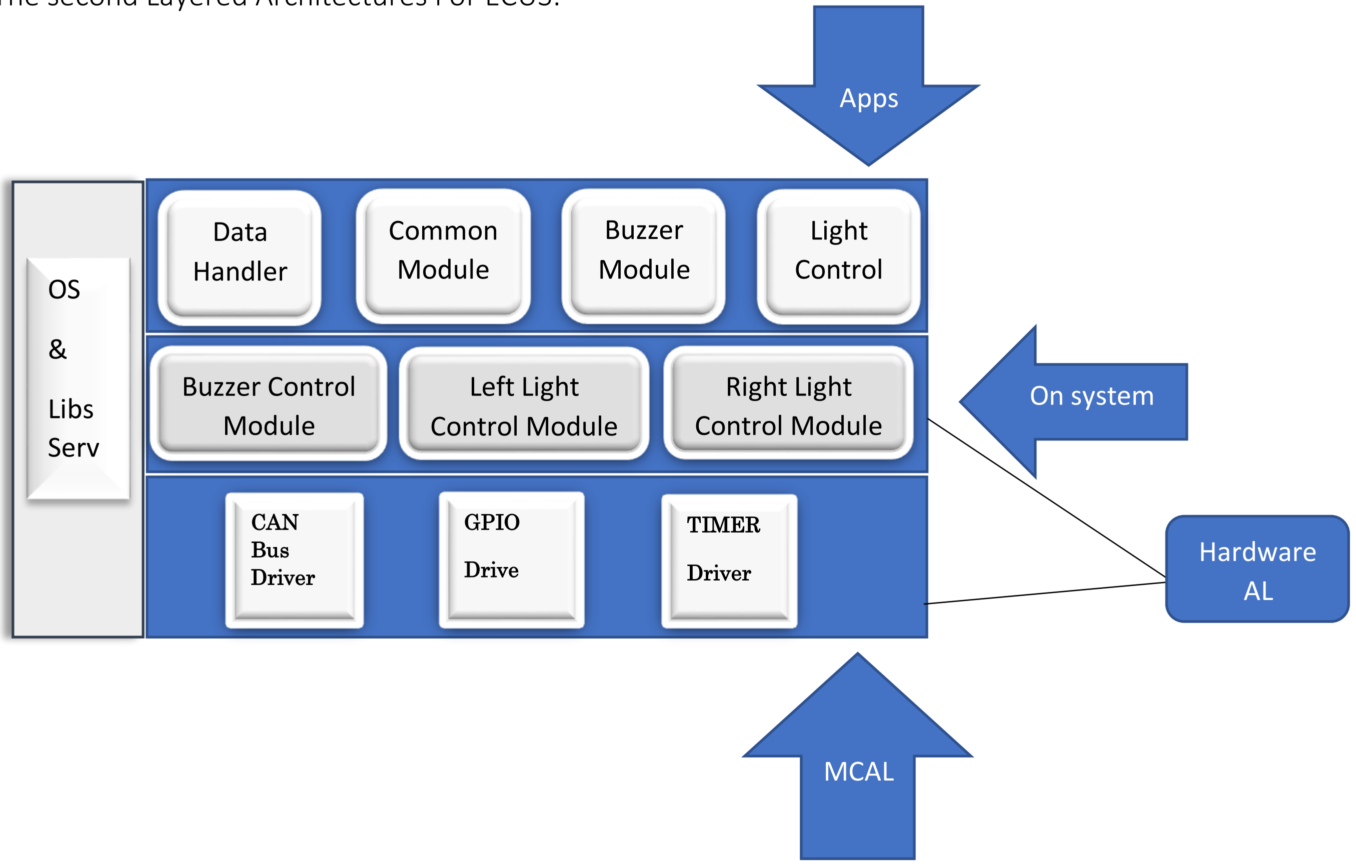
This PDF provides a comprehensive overview of the static design for an automotive door control system. It includes a detailed description of the layered architecture of the system, as well as the components and APIs of each layer. The purpose of this design is to illustrate the technical specifications and guidelines for implementing a reliable and efficient door control system in a vehicle. The information presented here will help stakeholders understand the system requirements and design considerations and provide a solid foundation for the implementation of the system.

The Layered Architectures For ECUS:

- The first Layered Architectures For ECUS:



- The second Layered Architectures For ECUS:



Then, we outline the various components and modules within the ECU abstraction layer and its accompanying low-level drivers. Our approach begins with the lower layers in order to establish a solid foundation for the higher-level ECU abstraction layer and the application.

- Low Layers:

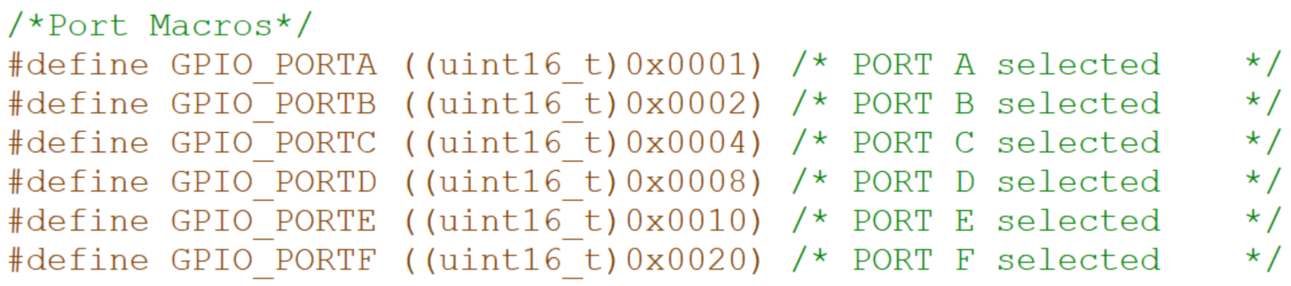
➔ Microcontroller Abstraction Layer:

- GPIO (General purpose input-output):

…describes the utilization of digital input/output in the ECU Abstraction Layer. The ECUAL layer will use it to communicate with the sensors and switches attached to the MCU. In order to control the operation of the GPIO module, full functional APIs must be provided for reading and writing data and controlling external interrupts on the pins.

API Type used for specifying the required PORT to control:

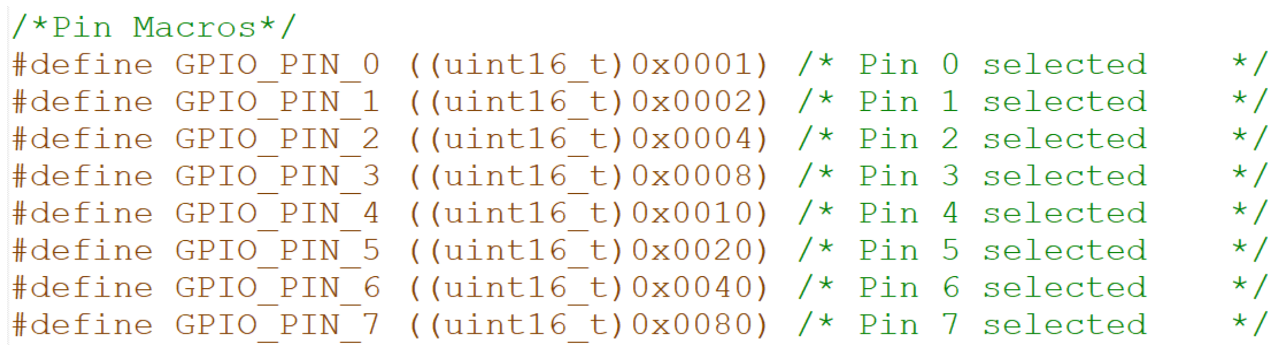
- Macro to define GPIO ports of ECU



```
/*Port Macros*/
#define GPIO_PORTA ((uint16_t)0x0001) /* PORT A selected    */
#define GPIO_PORTB ((uint16_t)0x0002) /* PORT B selected    */
#define GPIO_PORTC ((uint16_t)0x0004) /* PORT C selected    */
#define GPIO_PORTD ((uint16_t)0x0008) /* PORT D selected    */
#define GPIO_PORTE ((uint16_t)0x0010) /* PORT E selected    */
#define GPIO_PORTF ((uint16_t)0x0020) /* PORT F selected    */
```

API Type for Port I

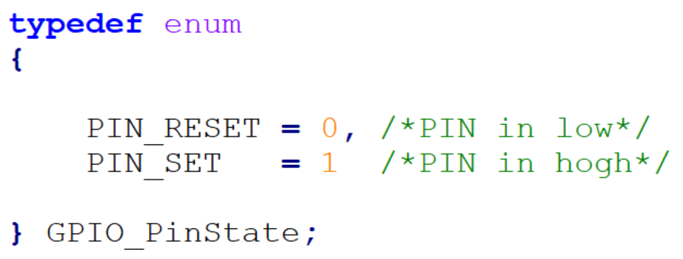
API Type used for specifying the required PIN to control:



```
/*Pin Macros*/
#define GPIO_PIN_0 ((uint16_t)0x0001) /* Pin 0 selected    */
#define GPIO_PIN_1 ((uint16_t)0x0002) /* Pin 1 selected    */
#define GPIO_PIN_2 ((uint16_t)0x0004) /* Pin 2 selected    */
#define GPIO_PIN_3 ((uint16_t)0x0008) /* Pin 3 selected    */
#define GPIO_PIN_4 ((uint16_t)0x0010) /* Pin 4 selected    */
#define GPIO_PIN_5 ((uint16_t)0x0020) /* Pin 5 selected    */
#define GPIO_PIN_6 ((uint16_t)0x0040) /* Pin 6 selected    */
#define GPIO_PIN_7 ((uint16_t)0x0080) /* Pin 7 selected    */
```

API Type for Pin I 1

API Type enum used to read and control the pins state:



```
typedef enum
{

    PIN_RESET = 0, /*PIN in low*/
    PIN_SET   = 1  /*PIN in hogh*/

} GPIO_PinState;
```

API Type used to struct the configuration parameters and passing to the initializing API:

```
typedef struct
{
    uint16_t Port; /* Specifies the GPIO Port to be configured.*/

    uint16_t Pin; /* Specifies the GPIO pins to be configured*/

    uint32_t Pull; /*Specifies the Pull-up or Pull-Down activation for the selected pins*/

    uint32_t Alternate; /*Peripheral to be connected to the selected pins*/
} GPIO_Init_Config;
```

API functions to initialize the DIO module and control operations:

```
 /*Initialize the GPIO
 @ to init the congigured pins
 */
void GPIO_Init(GPIO_Init_Config *pstr_Init);


/*Control functions
 @APIs to read,write and toggle pins on the ports
*/
GPIO_PinState GPIO_ReadPin(uint16_t *GPIOx, uint16_t GPIO_Pin);
void GPIO_WritePin(uint16_t GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);
void GPIO_TogglePin(uint16_t GPIOx, uint16_t GPIO_Pin);

/*Callack functions*/
void GPIO_EXTI_CallbackSet(uint16_t GPIOx,uint16_t GPIO_Pin,void (*EXTI_Callback)(void));
#endif /*End of file GPIO file*/
```

- GPT (General purpose timer):

According to HRS, the target microcontroller is equipped to connect to multiple sensors that use the timer module for timing management and synchronization of the communication bus, as it regularly sends trace data over the CAN bus.

This driver must offer APIs that utilize the hardware timers in the MCU and create precise time-based event triggers for a specified number of times, and an API for provisioning.

API Type used for initialization the channels:

```
#define TIM0 ((uint32_t)0X0000001)  /*Timer 0 sellected*/
#define TIM1 ((uint32_t)0X0000010)  /*Timer 1 sellected*/
#define TIM2 ((uint32_t)0X0000100)  /*Timer 2 sellected*/
#define TIM3 ((uint32_t)0X0001000)  /*Timer 3 sellected*/
#define TIM4 ((uint32_t)0X0010000)  /*Timer 4 sellected*/
#define TIM5 ((uint32_t)0X0100000)  /*Timer 5 sellected*/
```

API Type used to configure used perscaler:

```
#define TIMER_PRESCALER_1     1    /*PRESCALER Value 1 selected*/
#define TIMER_PRESCALER_8     8    /*PRESCALER Value 8 selected*/
#define TIMER_PRESCALER_64   64    /*PRESCALER Value 64 selected*/
#define TIMER_PRESCALER_256 256    /*PRESCALER Value 265 selected*/
```

API Type used to configure used clock type:

```
#define TIMER_CLOCK_INTERNAL        0 /*Internal timer is used*/
#define TIMER_CLOCK_EXTERNAL        1 /*External timer is used*/
```

API Type used to configure the modes:

```
typedef enum
{

    GPT_MODE_CONT,               /*Continues mode sellected*/
    GPT_MODE_ONESHOT,            /*one shot mode sellected*/
    GPT_MODE_SPEC                /*Spec mode sellected*/

} GPT_ModeType;
```

API Type used to configure return state:

```
typedef enum
{

  TIMER_OK,                 /*return state is done*/
  TIMER_ERROR               /*An error occured    */

} Timer_Status_t;
```

API Type used for struct the configuration parameters:

```c
typedef struct
{
  GPT_ModeType ChannelMode;                     /* configured mode*/
  uint8_t prescaler;                            /* configured prescaler */
  uint8_t clock_source;                         /* configured Timer type */
  uint32_t TickFrequency;                       /* configured frequency */
  void (*PeriodElapsedCallback)(void);          /*Set call back to Elapsed API */
} Timer_Config_t;
```

API functions used for initialization the driver and control operations:

```c
 /*Initialize the can*/
Timer_Status_t Timer_Init(Timer_Config_t* config, Timer_Callback_t callback);

/*Control functions */
Timer_Status_t GPT_Timer_Start(void);
Timer_Status_t GPT_Timer_Stop(void);
uint32_t GPT_GetTimeElapsed(TIM_TypeDef Channel);
uint32_t GPT_GetTimeRemaining(TIM_TypeDef Channel);

/*Callack functions*/
void GPT_Timer_Callback_t(void (*callback)(void));
```

- Input Capture Module:

The IC Driver must offer APIs to accurately measure the duration between rising and falling edges of the signals received from the sensors. This driver utilizes a timer unit within the target system. The ECUAL layer relies on this driver to perform its component APIs, making precise implementation crucial for obtaining accurate sensor data. In this section.

We detail the types and functions of the module APIs.

API Type used to configure return state:

```c
typedef enum
{

  IC_OK,                /*return state is done*/
  IC_ERROR              /*An error occured    */

} IC_Status_t;
```

API type used to identify periodicity:

```c
#define IC_OS_TICK_PERIOD_MS       10 /*Macro to configure the period of message tick*/
```

API type to struct the configuration parameters of the module and pass it to the initialization API function:

```
typedef struct
{

  uint8_t  IC_prescaler;     /*specifies the Input Capture Prescaler*/
  uint32_t IC_channel;       /* Specifies the channel of the input Capture*/
  uint32_t IC_selection;     /*Specifies the input*/
  uint32_t IC_polarity;      /*Specifies the active edge of the input signal*
  uint32_t IC_frequency_hz; /*Specifies the frequency*/

} IC_Config_t;
```

API functions to initialize the IC Module:

```
 /*Initialize the can*/
IC_Status_t ic_init(IC_Config_t *pstinit);
IC_Status_t ic_deinit(uint32_t IC_channel);
```

API functions to control the IC Module operation:

```
/*Control functions */
void ic_start_scheduler(void);
void ic_delay_ms(uint32_t delay_ms);
```

API functions to set Call back function to IC module:

```
/*Callack functions*/
void ic_register_os_tick_callback(void (*callback)(void));
```

- CAN bus module:

The first ECU is tasked with transmitting collected data from sensors on a CAN bus periodically. In the higher levels of the architecture, the communication module is defined, and it relies on the APIs of the CAN driver to carry out its functions. Hence, the driver APIs have been defined to meet the necessary communication specifications.

API type used to identify baud rate for CAN bus:

```
#define CAN_BAUD_RATE 500000                /*Macro to set the baud rate of CAN*/
```

API Type used to configure return state:

```
typedef enum
{

    CAN_OK,                         /*return state is done*/
    CAN_ERROR                       /*An error occured    */

} CanStatus_t;
```

API Type used to configure the module using CAN:

```
/*Configured Comp use CAN Bus*/
typedef enum
{

    DOOR_STATE,                       /*Door sensor uses CAN*/
    LIGHT_SWITCH_STATE,               /*Lght Switch uses CAN*/
    SPEED_STATE                       /*Speed sensor uses CAN*/

}can_message_type;
```

API Type used to struct the message sent by CAN:

```
typedef struct
{
    can_message_type type;      /*Configure the component using CAN*/
    uint8_t data;               /*The buffered data to be sent on CAN*/
    uint8_t len;                /*Length of the message sent on CAN*/

}can_message;
```

API functions to allow initialization and control operations:

```
 /*Initialize the can
 @initialize the CAN bus with the configured mode to receive the buffered data from the modules to transmit
 them to ECUS need the data
 */
void can_init(void);

/*Control functions
@
 @@ Control APIs to transmit and receive the data to and from the other ECUS
@*/
CanStatus_t Can_Transmit(can_message* pst_CAN_message);
CanStatus_t Can_Receive(can_message* pst_CAN_message);

/*Callack functions*/
void can_register_message_received_callback(void (*callback)(can_message *message));
```

ECU Abstraction Layer

In this section, we outline the APIs of the ECU Abstraction Layer (ECUAL). The Application Layer will rely directly on these APIs. As the components in this layer vary between the two ECUs in the system, we will separately define each group of components. We begin with the components of ECU 1.

ECU 1 Components:

1. Door Sensor:

   The sensor is expected to supply APIs for retrieving the state of the doors, as well as callbacks for any changes in the state.

   API type to identify the ID for the message:

```
#define DOOR_SENSOR_MESSAGE_ID                    0x100 /*Macro to define the ID of the sensor*/
```

   API type for retrieving the door state:

```
 /*
   @brief Door State structures definition
 */
 /*
 enum door state
 */
 typedef enum
{
    DOOR_CLOSED = 0, /* one door is opened        */
    DOOR_OPENED = 1  /* All the doors are closed  */

} DoorState;
```

   API Type used to struct the message sent by Door sensor:

```
/*
* @ struct of message to be buffered and sent by commu-control
*/
typedef struct {

    uint32_t message_id;      /*variable to carry the configured Id of the sensor*/
    DoorState door_state;     /*current door state*/
} DoorSensorMessage;
```

   API function to control operation and retrieve the state:

```
 /*Initialize the door sensor hardware here
    using DIO driver to set the required instruction for init
 */
void door_sensor_init(void);

/*Control functions
  @using read state to get the read of the written data on the sensor pins
  @sending the buffered data on CAN bus as a buffered message on a frame
*/
DoorState door_sensor_read_state(void);
void door_sensor_send_message(void);

/*Callack functions
  @call back function to send data back to lower layer
*/
void DoorSensor_Open_SetCallback (void (*OpenState_CallBack)(void));
void DoorSensor_Close_SetCallback(void (*CloseState_CallBack)(void));
```

2. Speed Sensor:

   This sensor is required to have API functions that can obtain measurements from the hardware sensor connected to the target. Additionally, the API should also be able to determine the state of the car's movement, with emphasis on the general state rather than a specific measurement for the speed.

   API type to identify the ID for the message:

```
#define SPEED_SENSOR_MESSAGE_ID 0x102 /*Macro to define the ID of the sensor*/
```

   API to identify the type of speed:

```
typedef uint16_t SpeedMeasure; /*data type to indentify speed rate */
```

   API Type used to struct the message sent by Speed sensor:

```
typedef struct {
    SpeedState speed_State;/* /*current speed state*/*/
    uint32_t message_id; /*variable to carry the configured Id of the sensor*/
    uint16_t speed;      /*current speed measurment*/

} SpeedSensorMessage;
```

   API function to control operation and retrieve measurements and state:

```
 /*Initialize the speed sensor*/
void speed_sensor_init(void);

/*Control functions : Read the speed of the car and return it*/
SpeedMeasure speed_sensor_read_speed(void);
void speed_sensor_send_message(void);
```

3. Light switch:

   This component will be tasked with reading the state of the light switch input. It will provide an API that returns the state of the switch when it is called. To ensure accurate results, the implementation will take measures to account for any potential noise on the pins by repeatedly reading the state of the pins over a specified number of periods. These periods can be controlled through the component's APIs.

   API type to identify the ID for the message:

```
#define LIGHT_SWITCG_MESSAGE_ID                     0x101 /*Macro to define the ID of the sensor*/
```

API type to identify the current state of the switch:

```
/*
  @brief light State structures definition
*/
/*
enum light switch
*/
typedef enum
{

    SWITCH_STATE_OFF = 1, /* LIGHT SWITCH IS NOT PRESSED     */
    SWITCH_STATE_ON = 0   /* LIGHT SWITCH IS PRESSED         */

} LightSwitchState;
```

API Type used to struct the message sent by light switch:

```
/*
* @ struct of message to be buffered and sent by commu-control
*/
typedef struct {

    LightSwitchState switch_state; /*current soor state*/
    uint32_t message_id;           /*variable to carry the configured Id of the switch*/

} LightSwitchMessage;
```

API function to initialize the component and get switches state as well as set the callback functions that called on changing the state:

```
 /*Initialize the light switch hardware here*/
void light_switch_init(void);


/*Control functions
  @using read state to get the read of the written data on the switch pins
  @sending the buffered data on CAN bus as a buffered message on a frame
*/
LightSwitchState light_switch_read_state(void); /*Read the state of the light switch*/
void light_switch_send_message(void);

/*Callack functions
  @call back function to send data back to lower layer
*/
void LightSwitch_On_SetCallback(void (*OnState_CallBack)(void));
void  LightSwitch Off SetCallback(void (*OffState CallBack)(void));
```

ECU 2 Components:

This ECU 2 will feature two light drivers, one each for the left and right lights, as well as a gate to control an alarm or buzzer. To simplify the control of these lights, they will be combined into one module component, with APIs provided to manage them both at once.

1. Light control

Here we define types and API functions to control on both of the right and left lights.

API types that used in selecting between the two drivers to control:

```
typedef enum
{
    Left_Light_Select ,     /* Left Light driver is selected to control    */
    Right_Light_Select ,    /* Right Light driver is selected to control   */
    Both_Light_Select       /* Both Light driver is selected to control    */

} LightSelect_t;
```

API type used to provide status of the performed operation:

```
typedef enum
{
     STATUS_OK,              /* Contol operation has successfully done   */
    STATUS_NOT_OK            /* Contol operation  has failed             */

} CTRLStatus_t;
```

API functions that used to initialization and control operations:

```
 /*Initialize the can*/
void LightCtrl_init(void);

/*Control functions */
CTRLStatus_t LightCtrl_Set_ON(LightSelect_t lightSel);
CTRLStatus_t LightCtrl_Set_OFF(LightSelect_t lightSel);
```

2. Alarm Control:

Here we define the APIs used to Control the Alarm operations.

API type used to identify the selected Buzzer operation:

```
typedef uint8_t BuzzerConfig_t;     /*Typedef to configure the sellected buzzer*/
```

API type used to identify the performed operation condition:

```
typedef enum
{

    BUZZER_OFF,                         /*Buzzer is turned off*/
    BUZZER_ON                           /*Buzzer is turned on*/

} BuzzerState_t;
```

API function to initialize the module and control operations:

```
 /*Initialize the can*/
void Buzzer_Init(BuzzerConfig_t *config);

/*Control functions */
void Buzzer_SetState(BuzzerState_t state);
```