DESIGN & IMPLEMENT A BCM

Youssef Ahmed Abbas Mohamed

SPRINTS

Contents

Project Introduction:

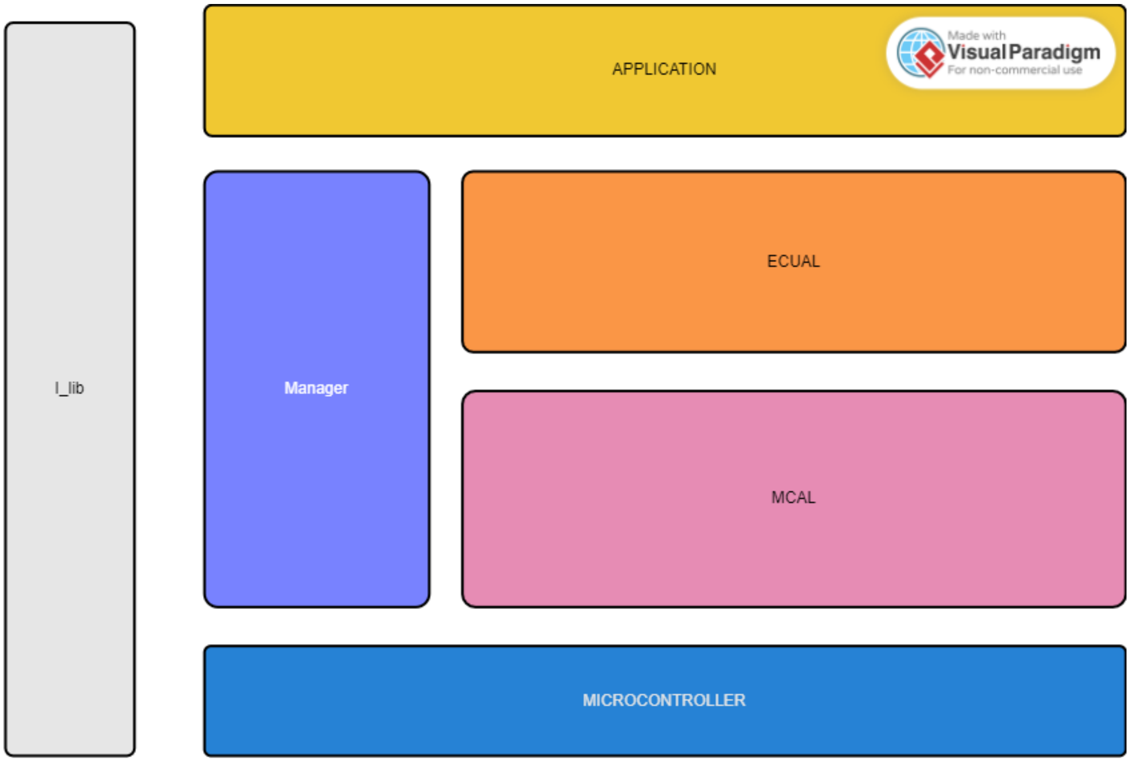
This project aims to implement a Communication Module (BCM) using the BCM Framework. The BCM is designed to facilitate data transmission and reception between different components or systems within a larger software application. It provides a flexible and efficient communication mechanism, supporting various communication protocols and data lengths up to 65535 bytes.

The implementation will be done using the C programming language, which offers low-level control and efficiency. Standard libraries and data structures will be utilized to ensure compatibility and optimal performance. The project will be developed and tested on an appropriate development environment, such as an Integrated Development Environment (IDE) or a text editor along with a compiler.

High Level Design:

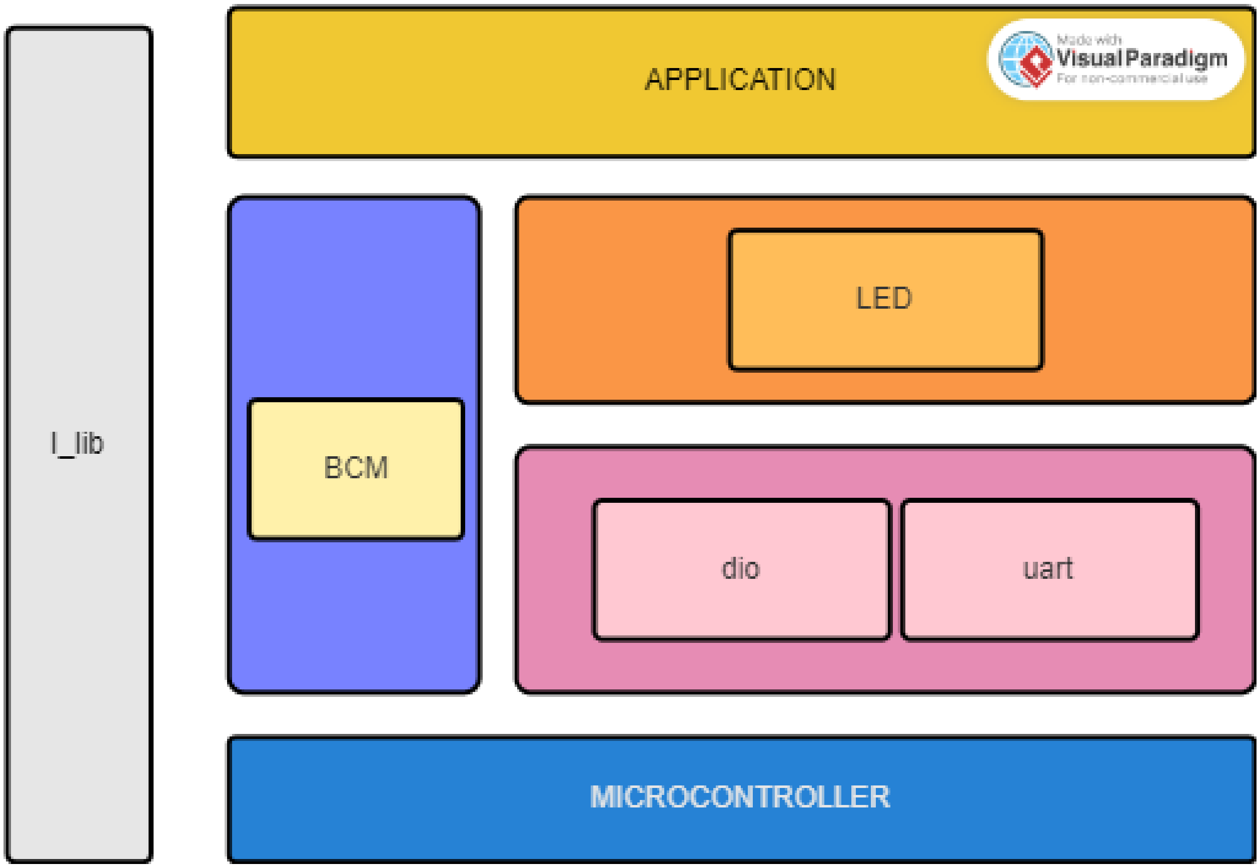
Layered architecture:

1. Application
2. Manager
3. ECUAL
4. MCAL
5. Microcontroller



Module Description

1. Application
2. ECUAL
    a. LED
3. Manager
    a. BCM
4. MCAL
    a. Dio
    b. Uart
5. Microcontroller



Driver Documentations

LED:

The module contains functions for initializing the LED, turning it on and off, and toggling its state.

To use this module, the **dio_interface.h** header file must be included. Additionally, the **str_dio_t** structure is used to configure the underlying digital input/output (DIO) pins associated with the LED.

Dependencies

- **dio_interface.h**: This header file defines the functions and data structures related to digital input/output (DIO) operations.

Data Types

enm_led_status_t

This enumerated type defines the possible states of the LED. It has the following values:

- **LED_ON**: Represents the LED being turned on (value: 1).
- **LED_OFF**: Represents the LED being turned off (value: 0).

str_led_t

This structure represents the LED and its associated properties. It contains the following members:

- **str_dio**: An instance of the **str_dio_t** structure that configures the DIO pins associated with the LED.
- **enm_led_status**: The current status of the LED, which can be either **LED_ON** or **LED_OFF**.

Functions

void LED_init(str_led_t* led)

This function initializes the LED by configuring the DIO pins and setting the initial LED status.

- **led**: A pointer to the **str_led_t** structure representing the LED to be initialized.

void LED_on(str_led_t* led)

This function turns the LED on by setting the appropriate DIO pin(s) to the active state.

- **led**: A pointer to the **str_led_t** structure representing the LED to be turned on.

void LED_off(str_led_t* led)

This function turns the LED off by setting the appropriate DIO pin(s) to the inactive state.

- **led**: A pointer to the **str_led_t** structure representing the LED to be turned off.

void LED_toggle(str_led_t* led)

This function toggles the state of the LED. If the LED is currently on, it will be turned off, and vice versa.

- **led**: A pointer to the **str_led_t** structure representing the LED to be toggled.

BCM Manager:

BCM (Communication Module) interface, which facilitates communication using different protocols such as UART, SPI, and I2C. The module includes functions for initializing and deinitializing the BCM, sending and receiving data, and executing periodic actions.

To use this module, the **std_types.h** header file must be included. The BCM operates on instances of the **str_bcm_instance_t** structure, which holds information about the communication protocol, instance ID, and specific protocol instance.

Dependencies

- **std_types.h**: This header file provides standard types used throughout the module.

Data Types

enm_cpo_t

This enumerated type defines the communication protocol options supported by the BCM. It has the following values:

- **BCM_PROTOCOL_UART**: Represents UART communication protocol (value: 0).
- **BCM_PROTOCOL_SPI**: Represents SPI communication protocol.
- **BCM_PROTOCOL_I2C**: Represents I2C communication protocol.
- **BCM_MAX_PROTOCOL**: Represents the maximum number of communication protocols supported.

enm_transiver_state_t

This enumerated type defines the states of the transceiver. It has the following values:

- **BCM_BUSY_FLAG**: Represents the transceiver being busy.
- **BCM_IDEL_FLAG**: Represents the transceiver being idle.

str_data_packet_t

This structure represents a data packet to be sent. It contains the following members:

- **ptr_data**: A pointer to the data buffer.
- **data_length**: The length of the data in the buffer.

str_rdata_packet_t

This structure represents a received data packet. It contains the following members:

- **ptr_data**: A pointer to the data buffer for storing received data.
- **data_length**: A pointer to a variable storing the length of the received data.

str_bcm_instance_t

This structure represents a BCM instance and its associated properties. It contains the following members:

- **bcm_instance_id**: The ID of the BCM instance.
- **protocol**: The communication protocol used by the instance (e.g., UART, SPI, I2C).
- **protocolInstance**: A pointer to the specific protocol instance.

Functions

enu_system_status_t bcm_init(str_bcm_instance_t* ptr_str_bcm_instance)

This function initializes the BCM module for a specific BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance to be initialized.

enu_system_status_t bcm_deinit(str_bcm_instance_t* ptr_str_bcm_instance)

This function deinitializes the BCM module for a specific BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance to be deinitialized.

enu_system_status_t bcm_send(str_bcm_instance_t* ptr_str_bcm_instance, uint8 *data)

This function sends a single byte of data over a specific BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance.
- **data**: A pointer to the data byte to be sent.

enu_system_status_t bcm_send_n(str_bcm_instance_t* ptr_str_bcm_instance, uint8* data, uint16 length)

This function sends multiple bytes of data over a specific BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance.
- **data**: A pointer to the data buffer to be sent.
- **length**: The length of the data buffer.

enu_system_status_t bcm_recive_n(str_bcm_instance_t* ptr_str_bcm_instance, uint8* data, uint16 *length)

This function receives multiple bytes of data over a specific BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance.
- **data**: A pointer to the buffer for storing received data.
- **length**: A pointer to a variable storing the maximum length of the received data. Upon completion, it will be updated with the actual length of the received data.

enu_system_status_t bcm_dispatcher(str_bcm_instance_t* ptr_str_bcm_instance,enm_transiver_state_t * state)

This function is a dispatcher that executes periodic actions and notifies events related to the BCM instance.

- **ptr_str_bcm_instance**: A pointer to the **str_bcm_instance_t** structure representing the BCM instance.
- **state**: A pointer to a variable storing the current state of the transceiver.

DIO:

Documentation: DIO (Digital Input/Output) Interface

Overview

DIO (Digital Input/Output) interface, which facilitates controlling and reading digital signals on specific pins and ports. The module includes functions for initializing pins, writing values to pins and ports, reading values from pins and ports, and toggling pin states.

To use this module, the **std_types.h** and **dio_private.h** header files must be included. The module defines enums for ports, pin values, pin directions, and DIO errors. It also includes a structure **str_dio_t** for representing a DIO pin.

Dependencies

- **std_types.h**: This header file provides standard types used throughout the module.
- **dio_private.h**: This header file provides private definitions and declarations for the DIO module.

Enums

enm_dio_port_t

This enumerated type defines the available ports for DIO pins. It has the following values:

- **PORT_A**: Represents Port A.
- **PORT_B**: Represents Port B.
- **PORT_C**: Represents Port C.
- **PORT_D**: Represents Port D.

enm_dio_value_t

This enumerated type defines the possible values for a DIO pin. It has the following values:

- **DIO_LOW**: Represents a low logic level (value: 0).
- **DIO_HIGH**: Represents a high logic level.

enm_dio_dir_t

This enumerated type defines the possible directions for a DIO pin. It has the following values:

- **DIO_IN**: Represents the input direction (value: 0).
- **DIO_OUT**: Represents the output direction.

enm_dio_error_t

This enumerated type defines the possible errors that can occur during DIO operations. It has the following values:

- **DIO_FAIL**: Represents a failure or error (value: 0).
- **DIO_SUCCESS**: Represents a successful operation.

Structures

str_dio_t

This structure represents a DIO pin and its associated properties. It contains the following members:

- **port**: The port to which the pin belongs (**enm_dio_port_t**).

- **pin**: The number of the pin within the port.

Functions

enm_dio_error_t dio_init(str_dio_t dio_pin, enm_dio_dir_t dir)

This function initializes a DIO pin with the specified direction.

- **dio_pin**: The **str_dio_t** structure representing the DIO pin to be initialized.
- **dir**: The desired direction for the pin (**DIO_IN** or **DIO_OUT**).

enm_dio_error_t dio_write_pin(str_dio_t dio_pin, enm_dio_value_t value)

This function writes a value to a DIO pin.

- **dio_pin**: The **str_dio_t** structure representing the DIO pin to be written to.
- **value**: The value to be written to the pin (**DIO_LOW** or **DIO_HIGH**).

enm_dio_error_t dio_toggle(str_dio_t dio_pin)

This function toggles the state of a DIO pin. If the pin is currently high, it will be set to low, and vice versa.

- **dio_pin**: The **str_dio_t** structure representing the DIO pin to be toggled.

enm_dio_error_t dio_read_pin(str_dio_t dio_pin, uint8 *value)

This function reads the value of a DIO pin and stores it in the provided variable.

- **dio_pin**: The **str_dio_t** structure representing the DIO pin to be read.
- **value**: A pointer to a variable where the pin value will be stored (**DIO_LOW** or **DIO_HIGH**).

enm_dio_error_t dio_write_port(enm_dio_port_t port, enm_dio_value_t value)

This function writes a value to the specified DIO port. The value will be applied to all pins of the port.

- **port**: The port to which the value will be written (**PORT_A**, **PORT_B**, **PORT_C**, or **PORT_D**).
- **value**: The value to be written to the port (**DIO_LOW** or **DIO_HIGH**).

enm_dio_error_t dio_read_port(enm_dio_port_t port, uint8 *data)

This function reads the value of a DIO port and stores it in the provided variable. The value represents the combined state of all pins in the port.

- **port**: The port to be read (**PORT_A**, **PORT_B**, **PORT_C**, or **PORT_D**).

- **data**: A pointer to a variable where the port value will be stored.

UART:

UART (Universal Asynchronous Receiver Transmitter) interface, which enables serial communication between devices. The module includes enums for various UART configurations and a structure **uart_config_t** to represent the UART configuration settings. Additionally, it defines functions for initializing the UART, writing and reading data, and enabling/disabling UART interrupts.

To use this module, the **std_types.h** header file must be included.

Enums

uart_receive_mode_t

This enumerated type defines the receive mode options for UART. It has the following values:

- **UART_RECEIVE_DISABLE**: Disable receive.
- **UART_RECEIVE_ENABLE**: Enable receive.

uart_transmit_mode_t

This enumerated type defines the transmit mode options for UART. It has the following values:

- **UART_TRANSMIT_DISABLE**: Disable transmit.
- **UART_TRANSMIT_ENABLE**: Enable transmit.

uart_udre_interrupt_mode_t

This enumerated type defines the interrupt mode options for UART's Data Register Empty (UDRE) interrupt. It has the following values:

- **UART_UDRE_INTERRUPT_DISABLE**: Disable the interrupt.
- **UART_UDRE_INTERRUPT_ENABLE**: Enable the interrupt.

uart_rxc_interrupt_mode_t

This enumerated type defines the interrupt mode options for UART's Receive Complete (RXC) interrupt. It has the following values:

- **UART_RXC_INTERRUPT_DISABLE**: Disable the interrupt.
- **UART_RXC_INTERRUPT_ENABLE**: Enable the interrupt.

uart_txc_interrupt_mode_t

This enumerated type defines the interrupt mode options for UART's Transmit Complete (TXC) interrupt. It has the following values:

- **UART_TXC_INTERRUPT_DISABLE**: Disable the interrupt.

- **UART_TXC_INTERRUPT_ENABLE**: Enable the interrupt.

uart_rx_mode_t

This enumerated type defines the receive mode options for UART. It has the following values:

- **UART_RX_DISABLE**: Disable receive.
- **UART_RX_ENABLE**: Enable receive.

uart_tx_mode_t

This enumerated type defines the transmit mode options for UART. It has the following values:

- **UART_TX_DISABLE**: Disable transmit.
- **UART_TX_ENABLE**: Enable transmit.

uart_speed_mode_t

This enumerated type defines the speed mode options for UART. It has the following values:

- **UART_SYNC_SPEED_MODE**: Synchronous mode.
- **UART_NORMAL_MODE**: Normal mode.
- **UART_DOUBLE_MODE**: Double speed mode.

uart_clock_polarity_t

This enumerated type defines the clock polarity options for UART. It has the following values:

- **UART_NO_CLOCK**: No clock in asynchronous mode.
- **UART_TXR_RXF**: Transmit rising, receive falling.
- **UART_TXF_RXR**: Transmit falling, receive rising.

uart_stop_mode_t

This enumerated type defines the stop bit options for UART. It has the following values:

- **UART_STOP_1_BIT**: One stop bit.
- **UART_STOP_2_BIT**: Two stop bits.

uart_parity_mode_t

This enumerated type defines the parity mode options for UART. It has the following values:

- **UART_PARITY_DISABLED**: Parity disabled.
- **UART_PARITY_EVEN**: Even parity mode.

- **UART_PARITY_ODD**: Odd parity mode.

uart_operating_mode_t

This enumerated type defines the operating mode options for UART. It has the following values:

- **UART_ASYNC_MODE**: Asynchronous mode.
- **UART_SYNC_MODE**: Synchronous mode.

uart_data_size_t

This enumerated type defines the data size options for UART. It has the following values:

- **UART_CS_5**: 5 bits length.
- **UART_CS_6**: 6 bits length.
- **UART_CS_7**: 7 bits length.
- **UART_CS_8**: 8 bits length.
- **UART_CS_9**: 9 bits length.

Structures

uart_config_t

This structure represents the configuration settings for the UART module. It contains the following members:

- **uart_mode**: The operating mode of the UART (asynchronous or synchronous).
- **uart_data_size**: The number of bits in a data frame.
- **uart_parity_mode**: The parity mode for error detection.
- **uart_stop_mode**: The number of stop bits.
- **uart_clock_polarity**: The clock polarity in asynchronous mode.
- **uart_speed_mode**: The speed mode (normal, double, or synchronous).
- **uart_receive_mode**: The receive mode (enable or disable).
- **uart_transmit_mode**: The transmit mode (enable or disable).
- **uart_udre_interrupt_mode**: The interrupt mode for Data Register Empty (enable or disable).
- **uart_rx_mode**: The receive mode (enable or disable).
- **uart_tx_mode**: The transmit mode (enable or disable).

- **uart_rxc_interrupt_mode**: The interrupt mode for Receive Complete (enable or disable).
- **uart_txc_interrupt_mode**: The interrupt mode for Transmit Complete (enable or disable).
- **uart_baudrate**: The desired baud rate for communication.

Function Prototypes

void uart_init(uart_config_t *uart_config)

This function initializes the UART module with the specified configuration.

- **uart_config**: A pointer to a **uart_config_t** structure containing the desired UART configuration settings.

void uart_write(uint16 *data)

This function writes a single character of data to the UART for transmission.

- **data**: A pointer to the data to be transmitted.

void uart_read(uint16 *data)

This function reads a single character of data from the UART.

- **data**: A pointer to a variable where the received data will be stored.

void uart_write_INT(void(*callback)(void))

This function enables interrupt-driven UART transmission. The provided callback function will be called when the UART is ready to transmit data.

- **callback**: A function pointer to the callback function that will be executed when the UART is ready to transmit data.

void uart_read_INT(void(*callback)(void))

This function enables interrupt-driven UART reception. The provided callback function will be called when data is received.

- **callback**: A function pointer to the callback function that will be executed when data is received.

void uart_udrei_enable(void)

This function enables the UART Data Register Empty (UDRE) interrupt.

void uart_udrei_disable(void)

This function disables the UART Data Register Empty (UDRE) interrupt.

void uart_rxci_enable(void)

This function enables the UART Receive Complete (RXC) interrupt.

void uart_rxci_disable(void)

This function disables the UART Receive Complete (RXC) interrupt.

void uart_txci_enable(void)

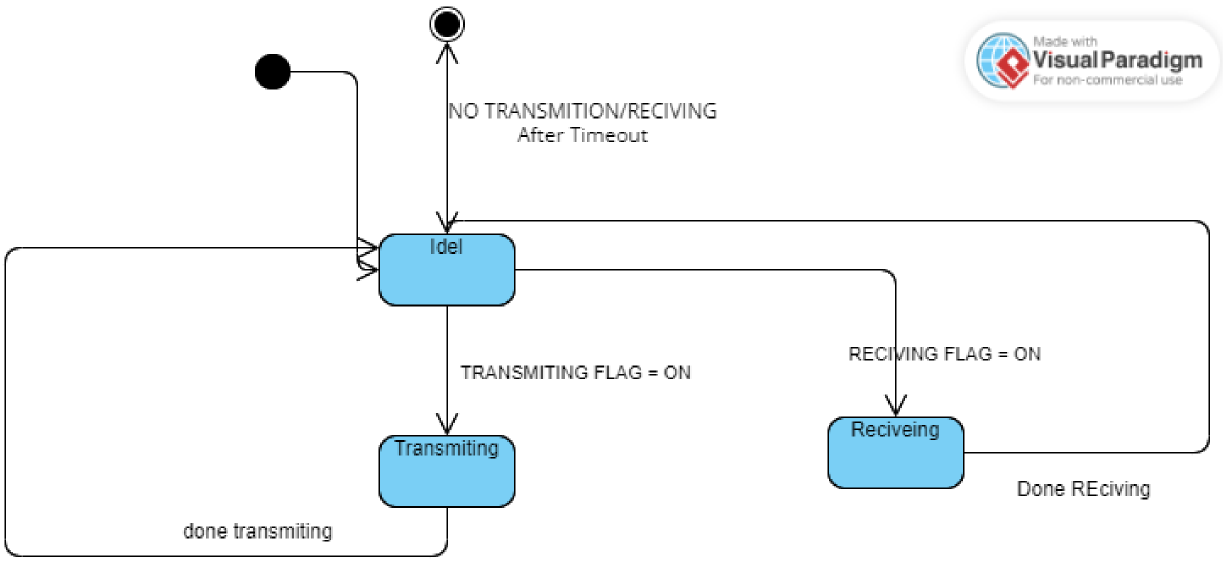
This function enables the UART Transmit Complete (TXC) interrupt.

void uart_txci_disable(void)

This function disables the UART Transmit Complete (TXC) interrupt.

UML:

State Machine:



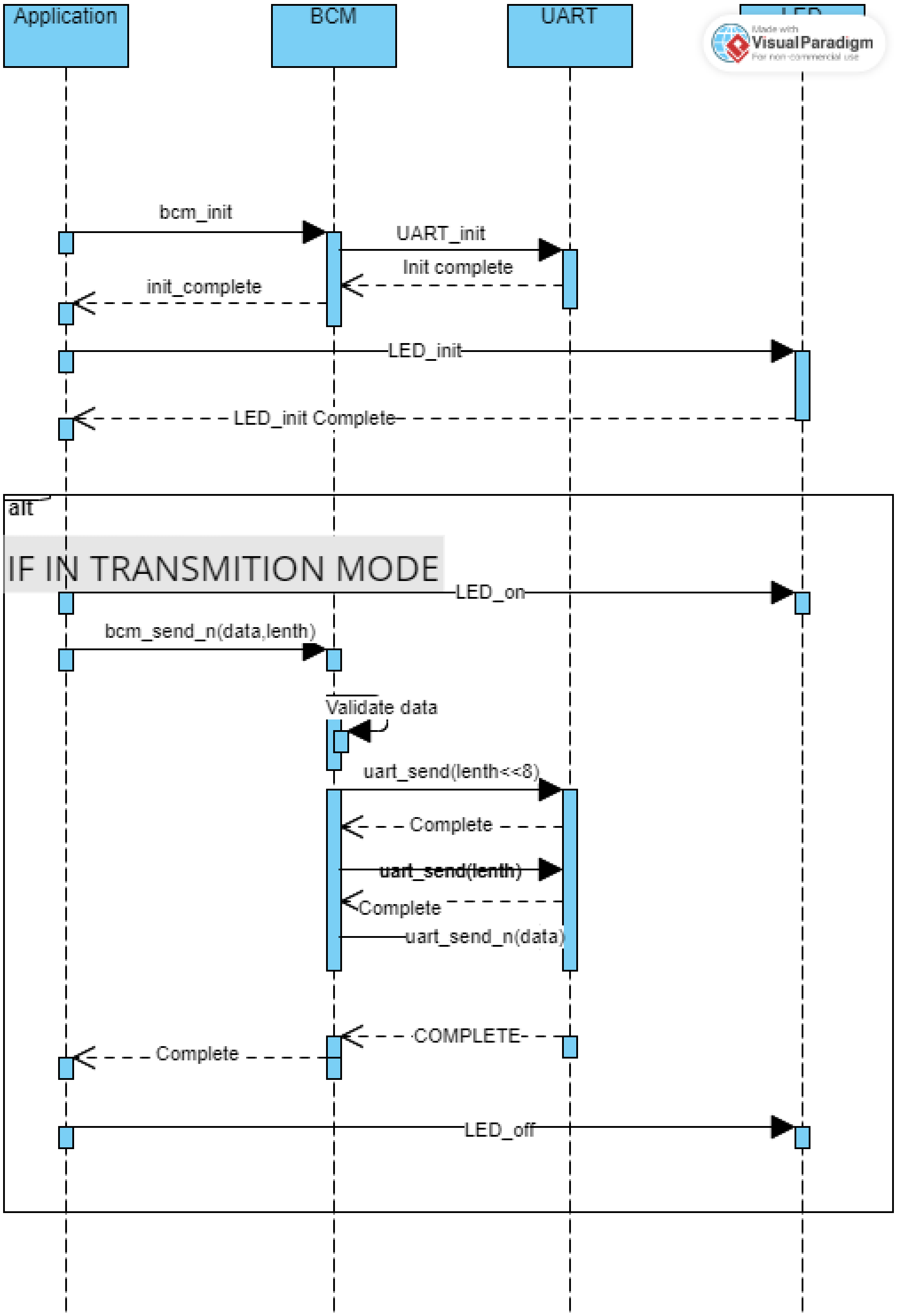
The state machine for the application consists of three states:

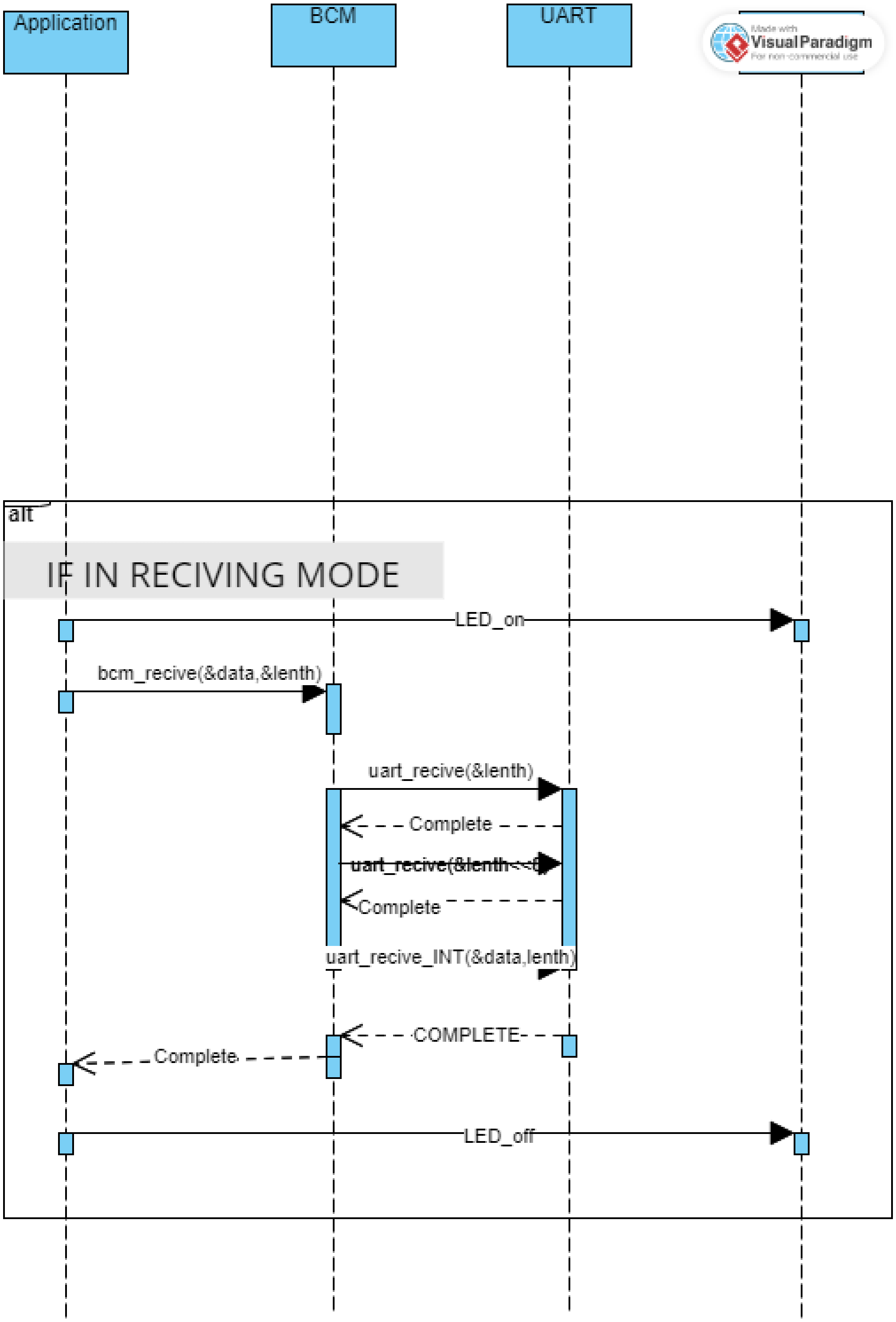
1. STATE_IDLE:
    - This state represents the idle state of the system.
    - When the state machine is in this state, it waits for the EVENT_START event to occur.
    - Upon receiving the EVENT_START event, it transitions to the STATE_TRANSMIT state.
2. STATE_TRANSMIT:

- This state represents the transmit state of the system.
- When the state machine is in this state, it performs the transmission operation using the BCM interface.
- After completing the transmission, it waits for the EVENT_TRANSMIT_COMPLETE event to occur.
- Upon receiving the EVENT_TRANSMIT_COMPLETE event, it transitions to the STATE_RECEIVE state.

3. STATE_RECEIVE:
    - This state represents the receive state of the system.
    - When the state machine is in this state, it performs the receive operation using the BCM interface.
    - After completing the receive operation, it waits for the EVENT_RECEIVE_COMPLETE event to occur.
    - Upon receiving the EVENT_RECEIVE_COMPLETE event, it transitions back to the STATE_TRANSMIT state.

Sequence Diagram




Application
BCM
UART
Made with Visual Paradigm For non-commercial use
alt
IF IN RECIVING MODE
LED_on
bcm_recive(&data,&lenth)
uart_recive(&lenth)
Complete
uart_recive(&lenth<<8
Complete
uart_recive_INT(&data,lenth)
COMPLETE
Complete
LED_off

Low Level Design:

Flowchart

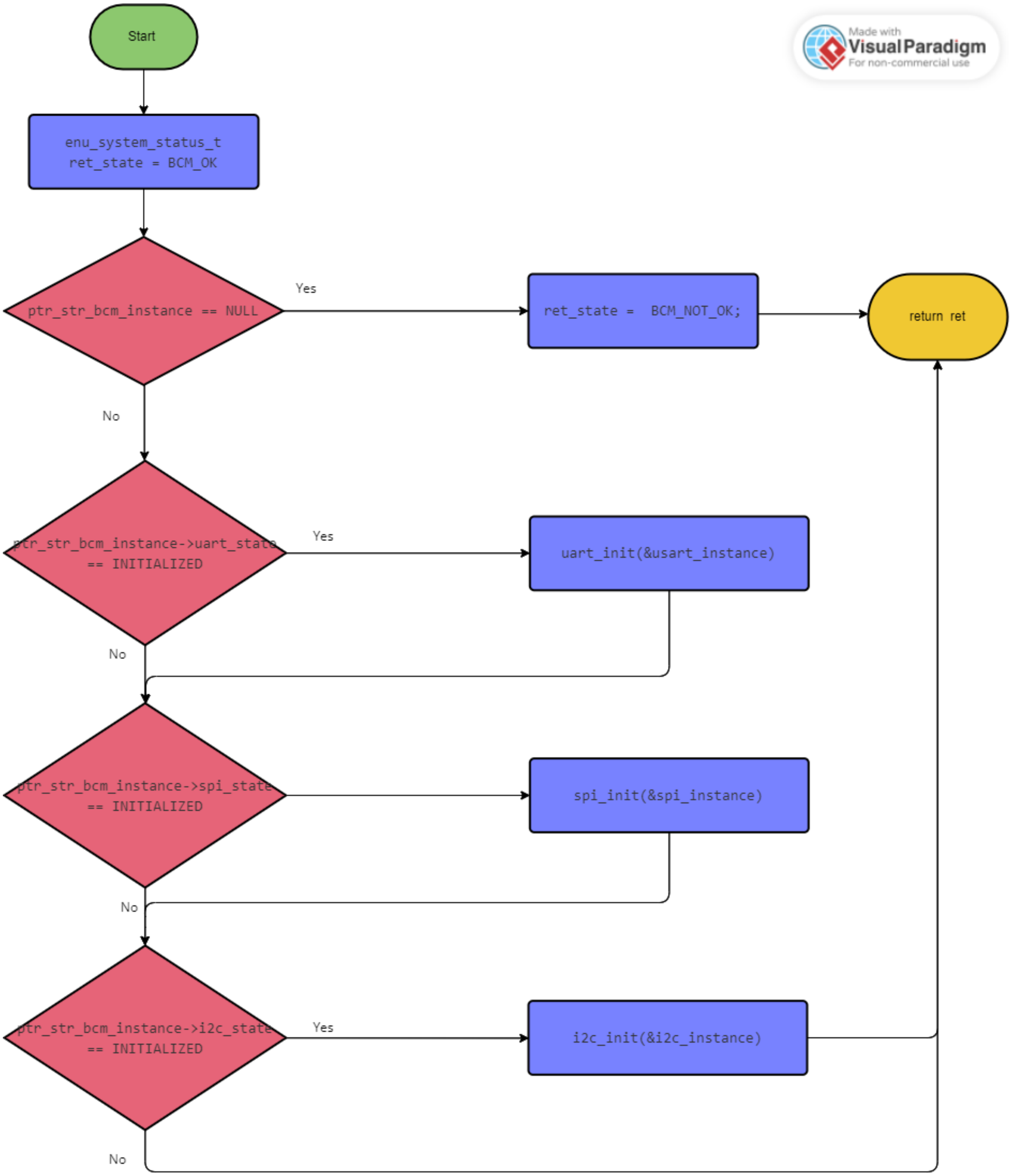

Start
enu_system_status_t ret_state = BCM_OK
ptr_str_bcm_instance == NULL
Yes
ret_state = BCM_NOT_OK;
return ret
No
ptr_str_bcm_instance->uart_state == INITIALIZED
Yes
uart_init(&usart_instance)
No
ptr_str_bcm_instance->spi_state == INITIALIZED
spi_init(&spi_instance)
No
ptr_str_bcm_instance->i2c_state == INITIALIZED
Yes
i2c_init(&i2c_instance)
No
Made with Visual Paradigm For non-commercial use

Figure 1 bcm_init.vpd

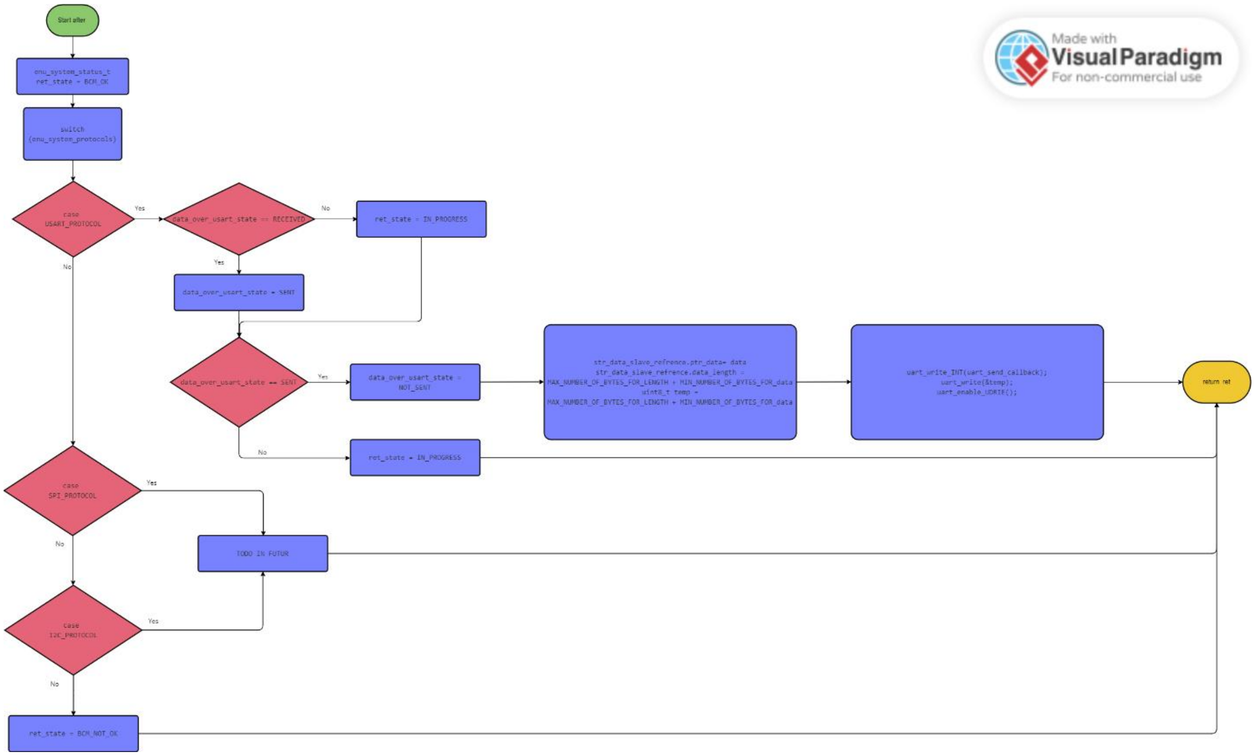


Figure 2 bcm_send.vpd

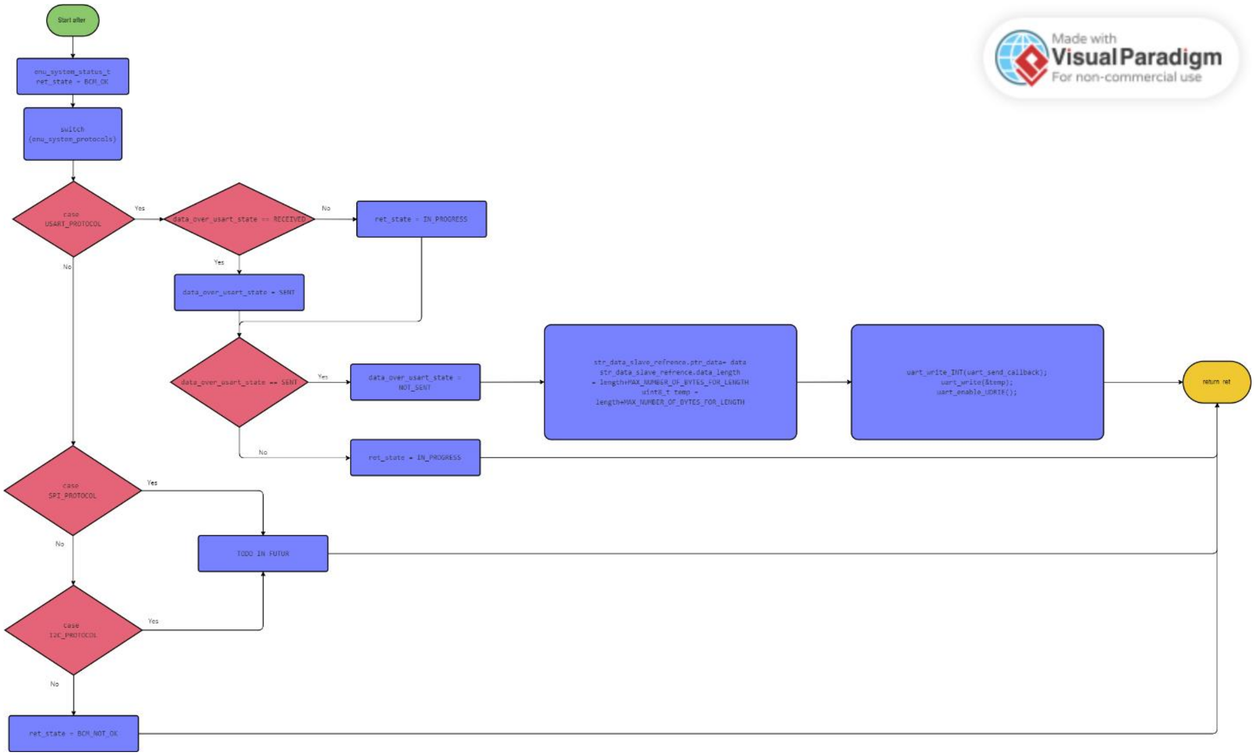


Figure 3 bcm_send_n.vpd

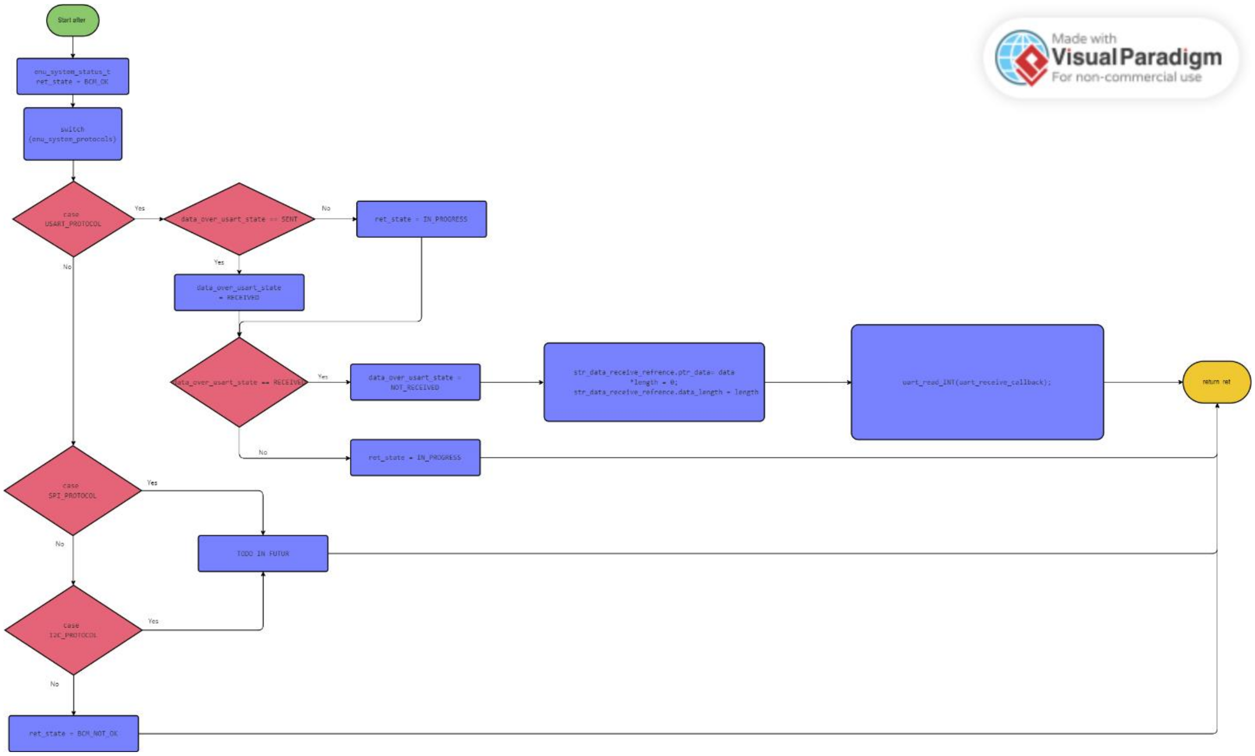


Figure 4 bcm_receive_n.vpd

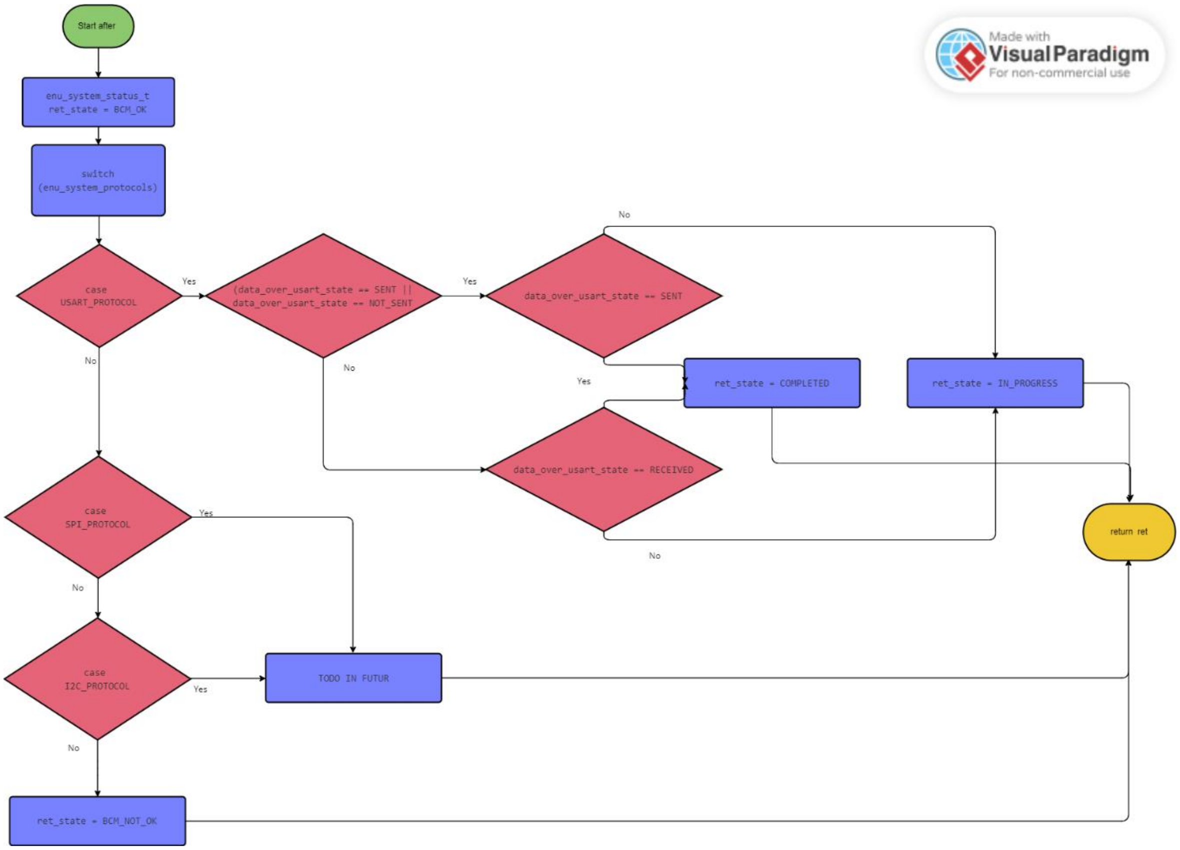


Figure 5 bcm_dispatcher.vpd

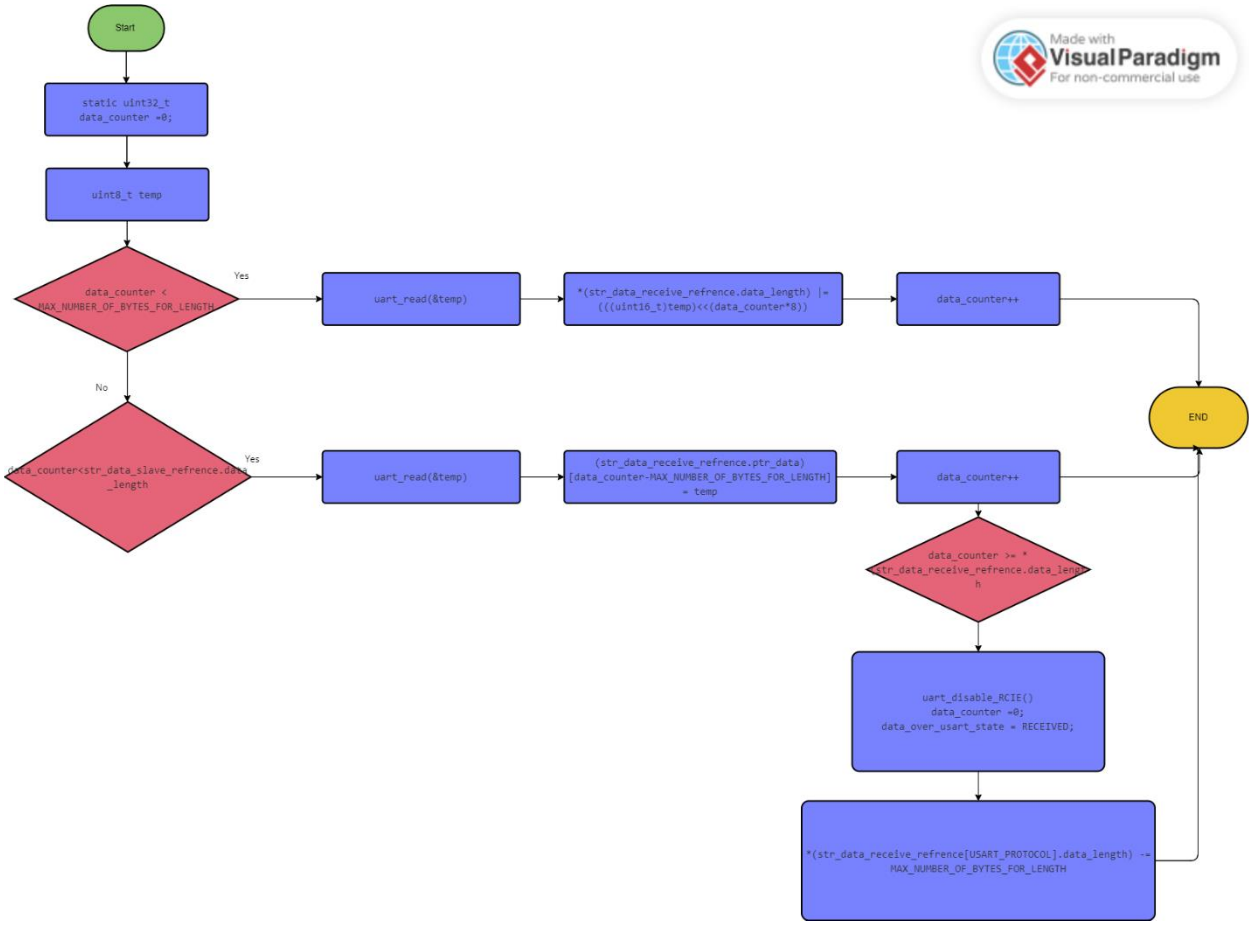

Made with
Visual Paradigm
For non-commercial use
Start
static uint32_t data_counter =0;
uint8_t temp
data_counter < MAX_NUMBER_OF_BYTES_FOR_LENGTH
Yes
uart_read(&temp)
*(str_data_receive_refrence.data_length) |= (((uint16_t)temp)<<(data_counter*8))
data_counter++
END
No
data_counter<str_data_slave_refrence.data_length
Yes
uart_read(&temp)
(str_data_receive_refrence.ptr_data)[data_counter-MAX_NUMBER_OF_BYTES_FOR_LENGTH] = temp
data_counter++
data_counter >= *(str_data_receive_refrence.data_length
h
uart_disable_RCIE()
data_counter =0;
data_over_usart_state = RECEIVED;
*(str_data_receive_refrence[USART_PROTOCOL].data_length) -= MAX_NUMBER_OF_BYTES_FOR_LENGTH

Figure 6 uart_receive_callback.vpd

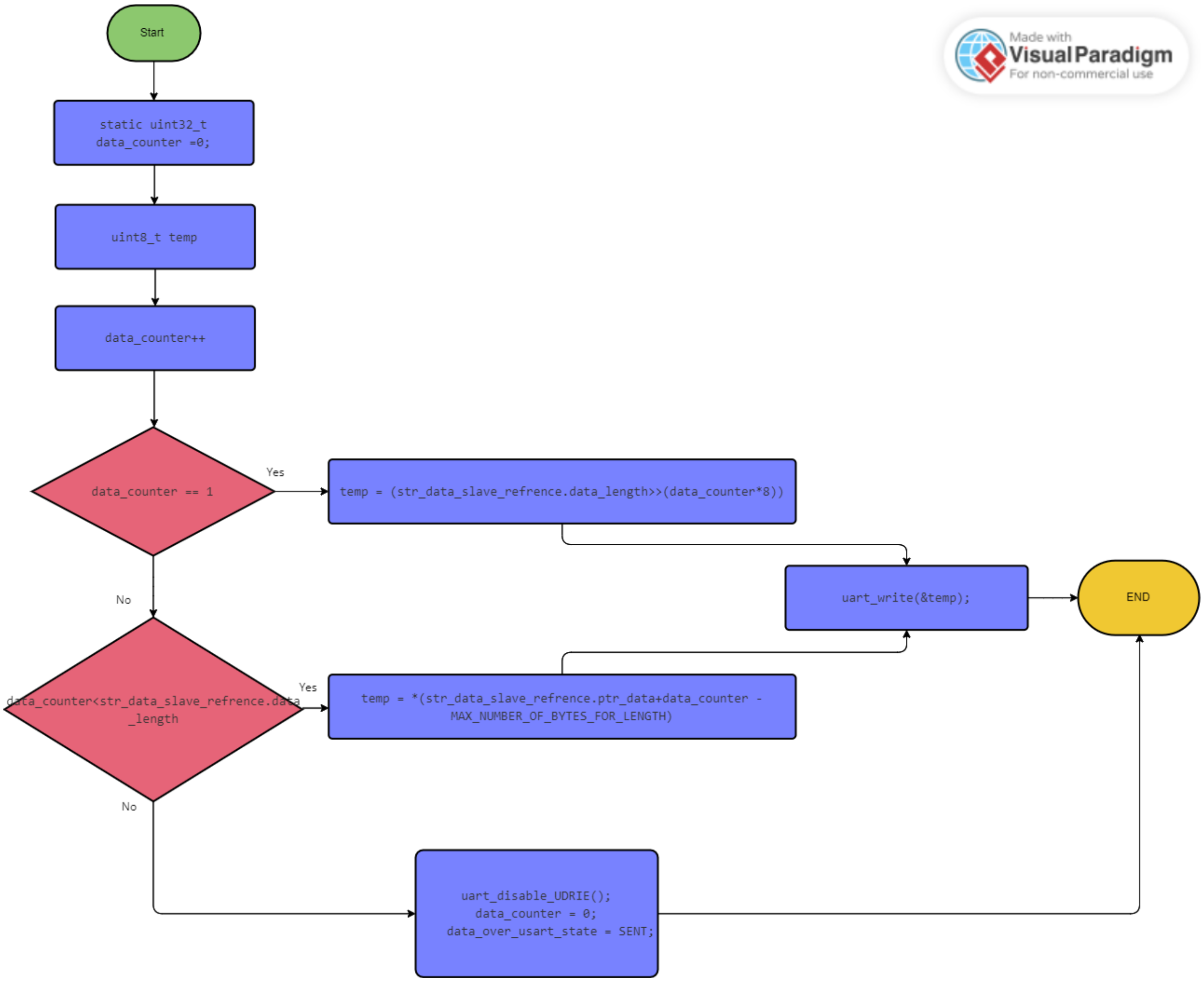

Made with Visual Paradigm For non-commercial use
Start
static uint32_t data_counter =0;
uint8_t temp
data_counter++
data_counter == 1
Yes
temp = (str_data_slave_refrence.data_length>>(data_counter*8))
No
uart_write(&temp);
END
data_counter<str_data_slave_refrence.data_length
Yes
temp = *(str_data_slave_refrence.ptr_data+data_counter - MAX_NUMBER_OF_BYTES_FOR_LENGTH)
No
uart_disable_UDRIE();
data_counter = 0;
data_over_usart_state = SENT;

Figure 7 uart_send_callback.vpd

Pre-compline

Application

```
#define BUFFER_SIZE          50
```

BCM

```
#define NUM_BCM_INSTANCES   3
#define LENGTH_BYTE_SIZE    2
#define DATA_BYTE_SIZE             1
```

STD

```
typedef unsigned char uint8;          /* 1-BYTE UNSIGNED DATA (0 - 255)
*/
```

```
typedef signed char   sint8;          /* 1-BYTE SIGNED DATA   (0 - 127) or (-1 - -128)
*/

typedef unsigned short int  uint16;   /* 2-BYTES UNSIGNED DATA   */
typedef signed short int    sint16;   /* 2-BYTES SIGNED DATA     */

typedef unsigned long int  uint32;     /* 4-BYTES UNSIGNED DATA  */
typedef signed long int    sint32;     /* 4-BYTES SIGNED DATA    */


typedef float  float32;                 /* 4-BYTES FLOATING DATA */
typedef double double64;                /* 8-BYTES FLOATING DATA */

#define NULL '\0'
```

BIT_MATH

```

/*macro to check if a bit is set*/
#define BIT_IS_SET(byte,bit_num) (byte & (1<<bit_num))

/*macro to check if bit is cleared*/
#define BIT_IS_CLEAR(byte,bit_num) (!(byte & (1<<bit_num)))
#define BIT_MASK        0x01

#define CLEAR_BIT(REG, BIT_POSN)            (REG &= (~(BIT_MASK << BIT_POSN)))  /*clear
a specific bit in reg*/
#define SET_BIT(REG, BIT_POSN)              (REG |= (BIT_MASK << BIT_POSN))     /*set a
specific bit in reg*/
#define TOGGLE_BIT(REG, BIT_POSN)           (REG ^= (BIT_MASK << BIT_POSN))
/*toggle specific bit in reg*/
#define READ_BIT(REG, BIT_POSN)             (((REG >> BIT_POSN) & BIT_MASK))    /*read
a specific bit in reg*/
```

Linking configuration

LED

```
typedef enum {
       LED_ON=1,
       LED_OFF = 0
}enm_led_status_t;
typedef struct{
       str_dio_t str_dio;
       enm_led_status_t enm_led_status;
}str_led_t;
```

BCM

```

// Communication protocol options
typedef enum {
       BCM_PROTOCOL_UART =0,
       BCM_PROTOCOL_SPI  ,
       BCM_PROTOCOL_I2C,
       BCM_MAX_PROTOCOL
} enm_cpo_t;

typedef enum {
       BCM_BUSY_FLAG=0,
       BCM_IDEL_FLAG
}enm_transiver_state_t;


typedef struct {
       uint8 * ptr_data;
       uint16 data_length;
}str_data_packet_t;
typedef struct {
       uint8 * ptr_data;
       uint16 * data_length;
}str_rdata_packet_t;


// BCM instance structure
typedef struct {
       uint8 bcm_instance_id;        // BCM instance ID
       enm_cpo_t protocol;               // Communication protocol (e.g., UART, SPI,
I2C)
       void* protocolInstance;         // Pointer to the specific protocol instance


} str_bcm_instance_t;



// System status enumeration
typedef enum {
       BCM_OK = 0,                     // Operation successful
       BCM_INVALID_INSTANCE,           // Invalid BCM instance ID
       BCM_ALREADY_INITIALIZED,        // BCM instance already initialized
       BCM_NOT_INITIALIZED,            // BCM instance not initialized
       BCM_INVALID_PROTOCOL,            // Invalid communication protocol
       BCM_INVALID_PARAMETER
} enu_system_status_t;
```

DIO

```
// define ports
typedef enum{
       PORT_A,
       PORT_B,
       PORT_C,
       PORT_D
}enm_dio_port_t;

// dio value
typedef enum {
       DIO_LOW = 0,
       DIO_HIGH
}enm_dio_value_t;

// dio direction
typedef enum {
       DIO_IN = 0,
       DIO_OUT
}enm_dio_dir_t;

// DIO Errors
typedef enum {
       DIO_FAIL=0,
       DIO_SUCCESS
}enm_dio_error_t;


typedef struct {
       enm_dio_port_t port;
       uint8 pin;
}str_dio_t;
```

UART

```
// Enums

typedef enum {
       UART_RECEIVE_DISABLE = 0,  // Disable receive
       UART_RECEIVE_ENABLE        // Enable receive
} uart_receive_mode_t;

typedef enum {
       UART_TRANSMIT_DISABLE = 0,  // Disable transmit
       UART_TRANSMIT_ENABLE        // Enable transmit
} uart_transmit_mode_t;

typedef enum {
       UART_UDRE_INTERRUPT_DISABLE = 0, // Disable interrupt
       UART_UDRE_INTERRUPT_ENABLE       // Enable interrupt
} uart_udre_interrupt_mode_t;
typedef enum {
       UART_RXC_INTERRUPT_DISABLE = 0,    // Disable RX
       UART_RXC_INTERRUPT_ENABLE          // Enable RX
} uart_rxc_interrupt_mode_t;

typedef enum {
       UART_TXC_INTERRUPT_DISABLE = 0,    // Disable TX
       UART_TXC_INTERRUPT_ENABLE          // Enable TX
} uart_txc_interrupt_mode_t;
typedef enum {
       UART_RX_DISABLE = 0,    // Disable RX
       UART_RX_ENABLE          // Enable RX
} uart_rx_mode_t;

typedef enum {
        UART_TX_DISABLE = 0,    // Disable TX
        UART_TX_ENABLE          // Enable TX
} uart_tx_mode_t;

typedef enum {
        UART_SYNC_SPEED_MODE = 0,   // Sync mode
        UART_NORMAL_MODE=0,           // Normal mode
        UART_DOUBLE_MODE            // Double speed
} uart_speed_mode_t;

typedef enum {
        UART_NO_CLOCK = 0,          // No clock in async mode
        UART_TXR_RXF,
        UART_TXF_RXR
} uart_clock_polarity_t;



typedef enum {
        UART_STOP_1_BIT = 0,           // One bit
        UART_STOP_2_BIT               // Two bits
} uart_stop_mode_t;

typedef enum {
        UART_PARITY_DISABLED = 0,   // No parity mode
        UART_PARITY_EVEN,           // Even parity mode
        UART_PARITY_ODD             // Odd parity mode
} uart_parity_mode_t;

typedef enum {
        UART_ASYNC_MODE = 0,        // Async mode
        UART_SYNC_MODE              // Sync mode
} uart_operating_mode_t;

typedef enum {
        UART_CS_5 = 0,              // 5 bits length
        UART_CS_6,                  // 6 bits length
        UART_CS_7,                  // 7 bits length
        UART_CS_8,                  // 8 bits length
        UART_CS_9= 7                // 9 bits length
} uart_data_size_t;

// Structures

typedef struct {
        uart_operating_mode_t uart_mode;
        uart_data_size_t uart_data_size;
        uart_parity_mode_t uart_parity_mode;
        uart_stop_mode_t uart_stop_mode;
        uart_clock_polarity_t uart_clock_polarity;
        uart_speed_mode_t uart_speed_mode;
        uart_receive_mode_t uart_receive_mode;
        uart_transmit_mode_t uart_transmit_mode;
        uart_udre_interrupt_mode_t uart_udre_interrupt_mode;
        uart_rx_mode_t uart_rx_mode;
        uart_tx_mode_t uart_tx_mode;
        uart_rxc_interrupt_mode_t uart_rxc_interrupt_mode;
        uart_txc_interrupt_mode_t uart_txc_interrupt_mode;
        uint16 uart_baudrate;
} uart_config_t;
```