BCM

BY: Momen Hassan

Contents

INTRODUCTION

The Basic Communication Manager (BCM) project aims to design a communication module that provides the capability to work with different serial communication protocols using an Interrupt Service Routine (ISR) for high throughput. The BCM module allows for sending and receiving data of up to 65535 bytes using various communication protocols.

This project consists of the implementation of the BCM driver, which includes initialization, deinitialization, sending data, receiving data, and a dispatcher function for periodic actions. The driver is designed to be generic, allowing for the integration of different communication protocols through function pointers.

The key features of the BCM driver include:

Support for various serial communication protocols.

Synchronous operation.

Non-reentrant design.

High throughput for data transmission and reception.

Capability to send and receive data of up to 65535 bytes.

The project also involves module testing to verify the functionality of the BCM driver. Test cases include sending and receiving a specific string between two microcontrollers (MCU_1 and MCU_2) and toggling LEDs upon successful transmission and reception.

The design of the BCM module follows a layered architecture, allowing for modular and scalable development. The module includes drivers for peripheral communication, such as UART, which provides the underlying functionality for data transmission and reception.

This project aims to provide a robust and efficient solution for handling serial communication in embedded systems. The BCM driver can be easily integrated into various applications and platforms, providing a flexible and reliable communication solution.

The design document accompanying this project will provide a detailed overview of the project's high-level architecture, module descriptions, drivers' documentation, UML diagrams, and sequence diagrams. Additionally, a video recording will be provided to further explain the design and implementation of the Basic Communication Manager module.

By implementing the BCM driver and conducting thorough testing, this project aims to deliver a reliable and efficient solution for serial communication in embedded systems, enabling seamless data transmission and reception across different communication protocols.

PROJECT COMPNENTS

1. 2x ATmega32 microcontroller
2. 4x LEDs (2 for each Controller)
3. Green LED: Transmit Complete
4. Blue LED: Receive Complete

SEYSTEM RECURMENT

1. bcm_Init using the below table. This function will initialize the corresponding serial communication protocol

Function Name	bcm_init
Syntax	bcm_status_t cm_init(bcm_instance_t* bcm_instance)
Sync/Async	Synchronous
Reentrancy	Non reentrant
Parameters (in)	Ptr_str_bcm_instance: address for bcm instance
Parameters (out)	None
Return	Enu_system_status_t

2. Implement **bcm_deinit** using the below table. This function will uninitialize thecorresponding BCM instance, (instance: is the communication channel)

Function Name	**bcm_deinit**
Syntax	bcm_status_t bcm_deinit(bcm_instance_t* bcm_instance)
Sync/Async	Synchronous
Reentrancy	Non reentrant
Parameters (in)	Ptr_str_bcm_instance: address for bcm instance
Parameters (out)	None
Return	Enu_system_status_t

3. **bcm_send** that will send only 1 byte of data over a specific BCM instance

Function Name	**bcm_send**
Syntax	bcm_status_t bcm_send(bcm_instance_t* bcm_instance, Uchar8_t data)
Sync/Async	Synchronous/Async (depends on comm protocol config)
Reentrancy	Re-entrant
Parameters (in)	bcm_instance, data
Parameters (out)	None
Return	`Enu_system_status_t`

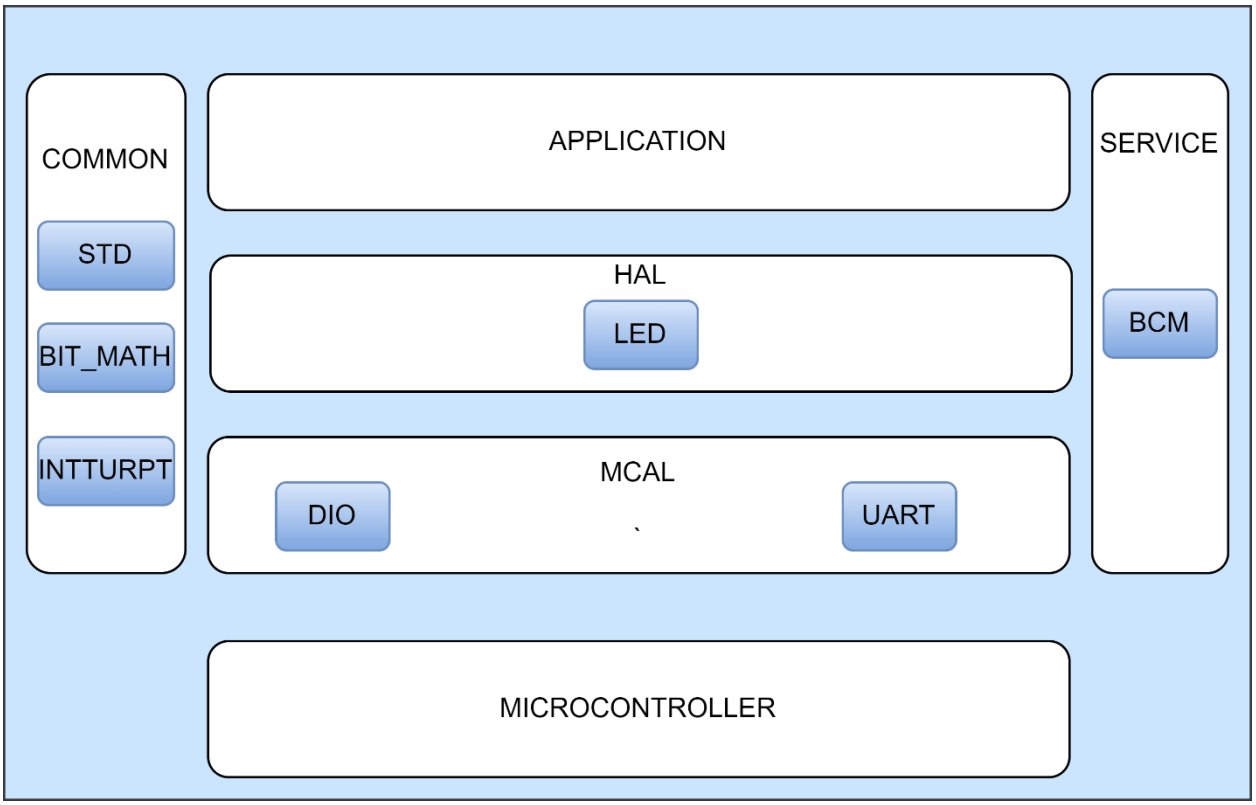
4. Implement **bcm_send_n** will send more than one byte with a length n over a specificBCM instance

Function Name	**bcm_send_n**
Syntax	bcm_status_t bcm_send_n(bcm_instance_t* bcm_instance, Uchar8_t* data, Uchar16_t length)
Sync/Async	Synchronous/Async (depends on comm protocol config)
Reentrancy	Re-entrant
Parameters (in)	bcm_instance, data, length
Parameters (out)	None
Return	`Enu_system_status_t`

5. Implement **bcm_dispatcher** will execute the periodic actions and notifies the userwith the needed events over a specific BCM instance

Function Name	**bcm_dispatcher**
Syntax	`bcm_status_t bcm_dispatcher(bcm_instance_t* bcm_instance)`
Sync/Async	Synchronous
Reentrancy	Non-reentrant
Parameters (in)	`ptr_str_bcm_instance: address for bcm instance`
Parameters (out)	None
Return	`Enu_system_status_t`

LAYERED ARCHITECTURE



MODULE, PERIPHERALS, & SUPPORTING DRIVERS DESCRIPTION

2.1.1. DIO (Digital Input/Output) Module

The *DIO* module is responsible for reading input signals from the system's sensors (such as buttons) and driving output signals to the system's actuators (such as *LEDs*). It provides a set of APIs to configure the direction and mode of each pin (input/output, pull-up/down resistor), read the state of an input pin, and set the state of an output pin.

2.1.2. UART Module

The *UART* Module (driver) for ATmega32 is a comprehensive solution for serial communication. It enables seamless integration of ATmega32 microcontrollers with externaldevices through the UART protocol. The module provides efficient and reliable data transmission, supporting both asynchronous and synchronous modes. With configurable baud rates and data formats, it accommodates various communication requirements. Its compact design and optimized code ensure minimal resource usage, allowing for efficient data transfer. Whether for interfacing with sensors, modules, or other microcontrollers, the UART Module for ATmega32 offers a robust and flexible solution for seamless serial communication.

2.1.3. LED Module

The *LED* Module (driver) for ATmega32 is a compact and versatile solution designed to control LEDs in various applications. With its support for ATmega32 microcontroller, it offersseamless integration and efficient LED management. The module provides easy-to-use functions for controlling individual LEDs, allowing for dynamic lighting effects and customization. Its compact design and optimized code ensure minimal resource utilization while delivering reliable and precise LED control.

2.1.4. BCM Module

The BCM Module (driver) for ATmega32 is a high-performance solution designed to handle diverse serial communication protocols. With its advanced capabilities, it seamlessly integrates with ATmega32 microcontroller, offering efficient and reliable data transfer. The module leverages Interrupt Service Routines (ISRs) to achieve maximum throughput, ensuring rapid and seamless communication. Its versatile design supports various serial communication protocols, making it adaptable to different applications. With meticulous hardware integration and software

DRIVERS' DOCUMENTATION

1. DIO

```
/*  Writes a voltage level to a pin of the DIO interface.
* Parameters|
*  [in] pin The pin to write the voltage level to
*  [in] volt The voltage level to write (HIGH or LOW
* Returns
*
*pin represents the pin to write the voltage level to, and volt represents the voltage
*level to write (HIGH or LOW).
*
*
* Parameters|          [in] pin The pin to write the voltage level to
*                      [in] volt The voltage level to write (HIGH or LOW).
* Returns
*       none
*/
void DIO_writepinn   (DIO_Pin_type   pin,DIO_PinVoltage_type volt)


/*  Reads the voltage level from a pin of the DIO interface.
*
*pin represents the pin to read the voltage level from, and volt is a pointer to store
*the read voltage level. The function reads the voltage level from the specified pin
*and stores it in the memory location pointed to by volt.
*
*
* Parameters|          [in] pin The pin to read the voltage level from.
*                      [out] volt Pointer to store the read voltage level.
* Returns
*       none
*/

void DIO_readpinn   (DIO_Pin_type   pin,DIO_PinVoltage_type *volt)

/*  Reads the voltage level from a pin of the DIO interface.
*
*pin represents the pin to read the voltage level from, and volt is a pointer to store
*the read voltage level. The function reads the voltage level from the specified pin
*and stores it in the memory location pointed to by volt.
*
* Parameters|          [in] pin The pin to toggle.
* Returns
*     none
*/
void DIO_togllepin (DIO_Pin_type   pin)
```

```
/*   Initializes a pin of the DIO interface with a given status.
*
*The function DIO_initpinn initializes a pin of the DIO interface with a given status.
*It takes two parameters: pin, which represents the pin number, and status, which
*represents the desired status of the pin (OUTPUT, INFREE, or INPULL).
*
 *  Parameters
 *    [in] pin The The pin number.
 *  [in] status The status of the pin (OUTPUT, INFREE, or INPULL).
* Returns
*  none
 */
void DIO_initpinn   (DIO_Pin_type   pin,DIO_PinStatus_type status)
```

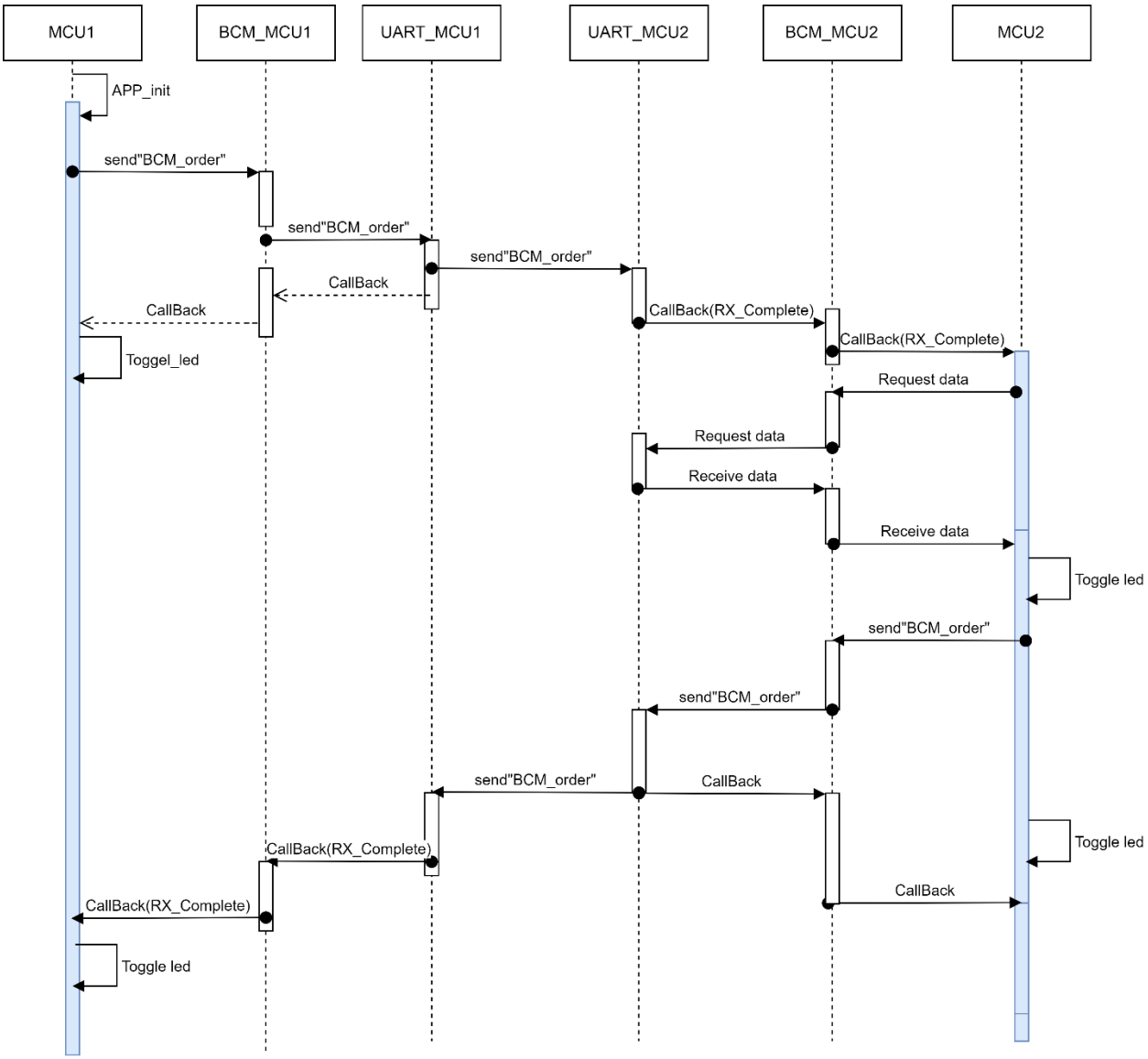
2. UART

```
/*   Initializes the UART interface.*
*
 *  Parameters
 *    [in] none
* Returns EN_UART_status_t indicating the status of the initialization.
*  none
 */
EN_UART_status_t UART_init (void)
```

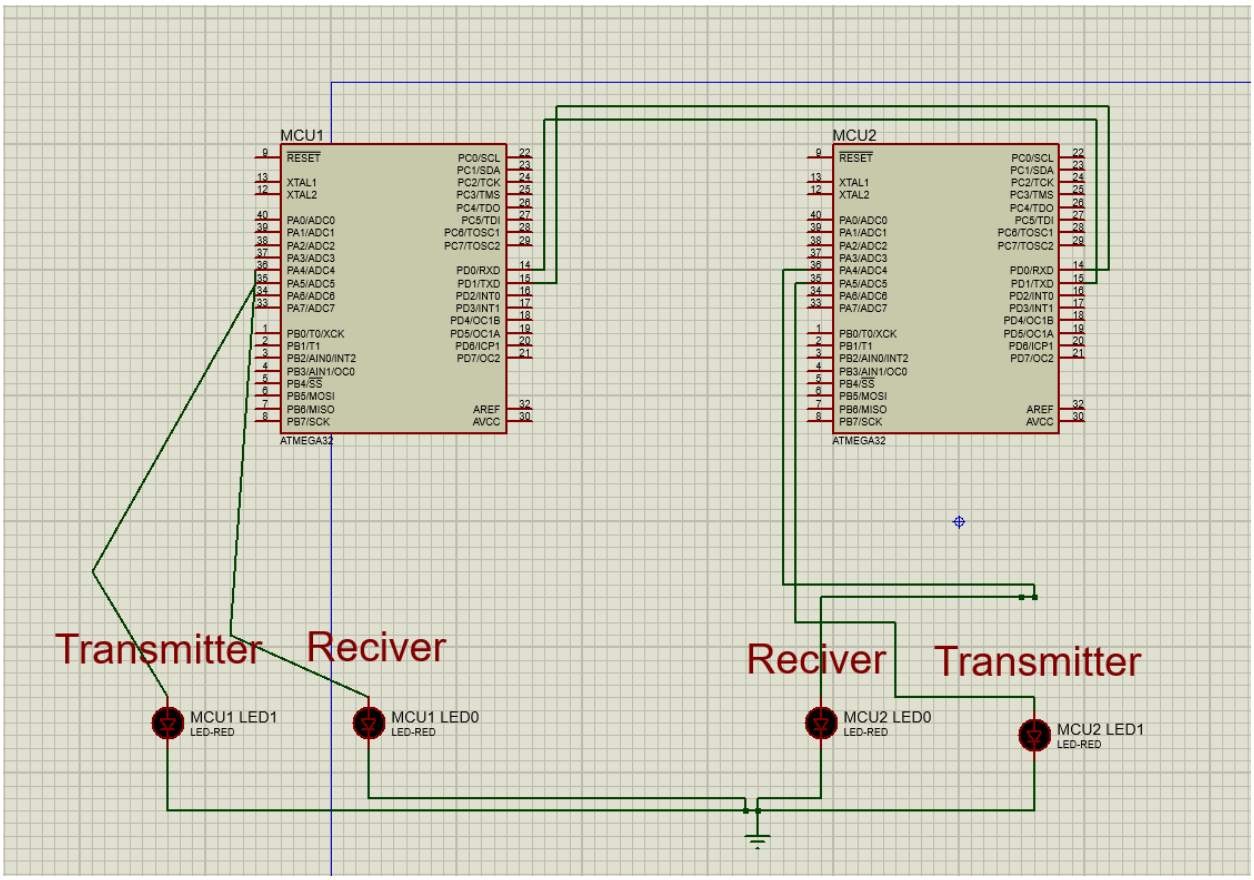
```
/*   Deinitializes the UART interface.
*
 *  Parameters
 *    [in] none
* Returns EN_UART_status_t indicating the status of the DE initialization.
*  none
 */
EN_UART_status_t UART_deinit (void)
```

```
/*   Sends a single byte over the UART interface..
*
*  Parameters
*    [in] Uchar8_t u8_a_byte - The byte to be sent
* Returns EN_UART_status_t indicating the status of the send operation.
*  none
 */
EN_UART_status_t UART_sendByte (Uchar8_t u8_a_byte)

/*   Sends a string of characters over the UART interface...
*
*  Parameters
*    [in] Uchar8_t *str - Pointer to the string to be sent. Uchar16_t u16_arr_size - Size of the string to be sent.
* Returns EN_UART_status_t indicating the status of the send operation.
*  none
 */
EN_UART_status_t UART_sendString (Uchar8_t * str , Uchar16_t u16_arr_size)

/*   Receives data over the UART interface..
*
*  Parameters
*    [in] Uchar16_t u16_arr_size - Size of the data to be received.
* Returns void.
*  none
 */
void uart_recive(Uchar16_t  u16_arr_size)
```

SEQUENCE DIGRAM



PROJECT SCHMETIC



CONFIGRATION

1. Dio

```
typedef enum{
    PA=0,
    PB,
    PC,
    PD
}DIO_Port_type;

typedef enum{
    OUTPUT,
    INFREE,
    INPULL
}DIO_PinStatus_type;

typedef enum{
    LOW=0,
    HIGH,
}DIO_PinVoltage_type;


typedef enum{
    PINA0=0,
    PINA1=1,
    PINA2,
    PINA3,
    PINA4,
    PINA5,
    PINA6,
    PINA7,
    PINB0,
    PINB1,
    PINB2,
    PINB3,
    PINB4,
    PINB5,
    PINB6,
    PINB7,
    PINC0,
    PINC1,
    PINC2,
    PINC3,
    PINC4,
    PINC5,
    PINC6,
    PINC7,
    PIND0,
    PIND1,
    PIND2,
    PIND3,
    PIND4,
    PIND5,
    PIND6,
    PIND7,
    TOTAL_PINS
}DIO_Pin_type;
```

55 %

2. uart

```
 // options for internal clock speed
typedef enum EN_FCPU_SELECTION{
    FCPU1M=0,
    FCPU2M,
    FCPU4M,
    FCPU8M,

}EN_FCPU_SELECTION;

// BAUD rate options
typedef enum EN_BAUDRATE_SELECTION{
    BAUDRATE_2400=0,
    BAUDRATE_4800,
    BAUDRATE_9600,
    BAUDRATE_14400,


}EN_BAUDRATE_SELECTION;
/* USART MODE :
1-USART_ASYNC_MODE
2-USART_SYNC_MODE
*/
typedef enum EN_USART_SET_MODE {
    USART_ASYNC_MODE = 0 ,
    USART_SYNC_MODE
}EN_USART_SET_MODE;

/* USART PARITY OPTIONS :
1-USART_DIS_PARITY
2-USART_ODD_PARITY
3-USART_EVEN_PARITY
*/
typedef enum EN_USART_SET_PARITY_MODE{
    USART_DIS_PARITY = 0 ,
    USART_ODD_PARITY,
    USART_EVEN_PARITY
}EN_USART_SET_PARITY_MODE;

/* USART STOP BIT OPTIONS :
1-USART_ONE_STOP_BIT
2-USART_TWO_STOP_BITS
*/
typedef enum EN_USART_SET_STOP_BIT {
 USART_ONE_STOP_BIT = 0 ,
 USART_TWO_STOP_BITS
}EN_USART_SET_STOP_BIT;
```