An Implementation of an IR Transceiver using a STM32F407 Microcontroller

James deLeon
Department of Electrical Engineering, The University of Texas at Arlington
416 S. Yates St., Arlington, TX, 76010
james.deleon@mavs.uta.edu

***Abstract*—The STM32 category of 32-bit microcontrollers offer a wide range of capabilities for building applications on a high and low level, where unlike other families of microcontrollers, STM32 microcontrollers allow for intuitive interaction between the processor's programmable logic and peripheral registers. An implemented IR transceiver will utilize the USART/UART protocol of a STM32F407 microcontroller, where the USART peripherals will be defined from the ground up. This report will outline the key functionalities of the IR transceiver implementation, along with the corresponding code that allows for a USART connection to be established on a 32-bit register level.**

I. Introduction

The STM32F407 is a discovery board used commonly for prototype development with its 32-bit ARM Cortex processor. References for the MCU-specific microcontroller, specifically regarding the STM32FXXX category of MCU, indicate the registers necessary for peripheral-based applications; the peripherals focused on this report are the GPIO and USART peripherals. Specifically, the GPIO peripherals are necessary for IO mechanisms established by the user, and the USART peripherals being necessary for universal synchronous/asynchronous communication between the MCU and external signaling components.

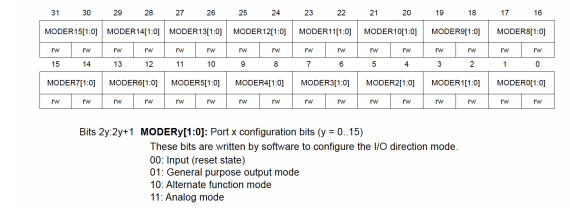
This report will focus on an implementation that utilizes USART communication through the STM32F407 MCU via an IR transceiver, where IR transmission is activated through keypad GPIO inputs and thus received via a GPIO-initialized LCD interface. Ultimately, keypad character inputs will be received as output characters on the LCD screen. This particular implementation will focus only on one STM32F407 MCU for USART TX/RX functions; it is important to note that this implementation is functional with more than one MCU connected via the same USART TX/RX bus.

The IR transceiver itself is an externally prototyped circuit that utilizes an IR LED transmitter and a VS1838 IR receiver ancillary to a logically inverted NE555P timer pulse established at approximately 38 kHz.

A. GPIO Peripheral

The STM32F407 is equipped with 11 GPIO ports (GPIOA-K) that, for implementation purposes, can interact with external terminals of the microcontroller board. Each GPIO port is configurable based on their corresponding address within the MCU, such that with the initialization of their corresponding peripheral clock, GPIO pins are configurable to read input pulses, transmit output pulses, and for implementing the USART communication protocol within the MCU.

Observing Fig. 1, GPIO registers can be configured to operate under specific modes, especially modes dealing with input, output and alternate functions. Alternate functions for the GPIO peripheral can configure TX/RX mechanisms for the USART/UART protocol, which are exclusive to the GPIO's alternate function mode #7.

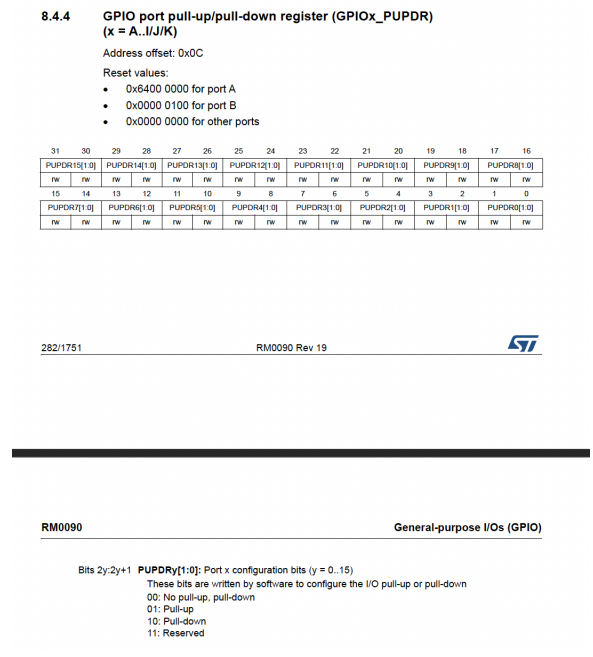


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Fig. 1. GPIO Mode Register Configuration

Otherwise, the LCD and keypad interfaces will be assigned to GPIO pins that are configured as either inputs or outputs. Input and output configurations can opt to instantiate embedded pull-up or pull-down resistors to assist in any floating point signal errors. Fig. 2 shows that a pull-up/down register can be configured for any GPIO peripheral initialized. It shall be explained further how the keypad interface will utilize the MCU's embedded PUPD configurations.



8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..I/J/K)

Address offset: 0x0C

Reset values:

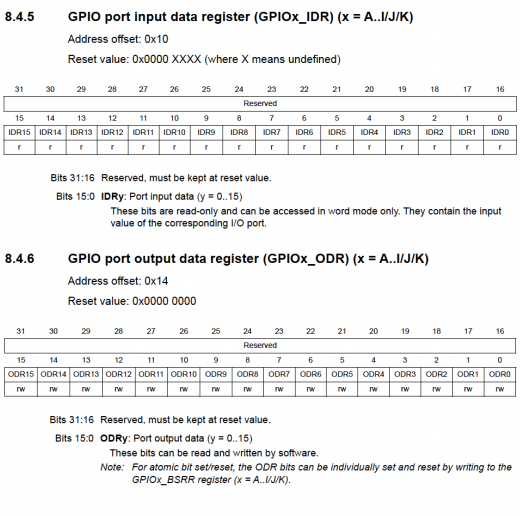
- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Fig. 2. GPIO PUPD Register Configuration

Finally, once the GPIO peripherals are configured, sending and receiving data is performed via the MCU's GPIO input/output data registers, as shown in Fig. 3. The data registers shown correspond to a GPIO pin for a specific configured port.



8.4.5 GPIO port input data register (GPIOx_IDR) (x = A..I/J/K)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy**: Port input data (y = 0..15)
These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

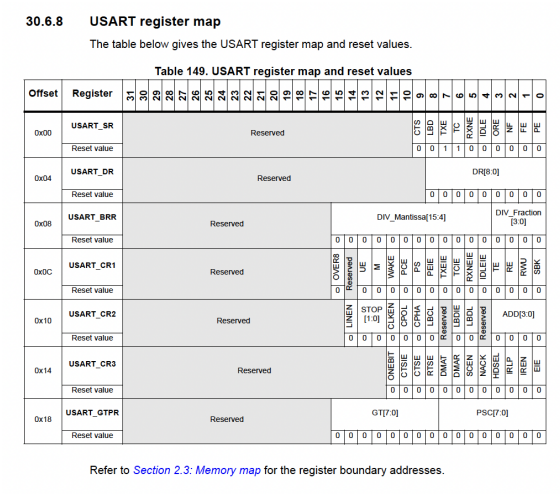
Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)
These bits can be read and written by software.
Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).

Fig. 3. GPIO IO Data Register Configuration

B. USART/UART Peripheral

The STM32F407 is equipped with 6 USART/UART ports that, for implementation purposes, can interact with external terminals of the microcontroller board through the GPIO's alternate functions. Specific baud rates calculated by the MCU's RCC peripheral allow for a wide range of data exchange frequencies. For this implementation, the slowest setting will be configured across USART2 and USART3, where USART2 will handle data transmission and USART3 will handle data reception. Fig. 4 below shows the mapping of USART registers, where the data register (DR) handles communication, the select register (SR) handles switching between a transmitting/receiving state, and the baud rate register (BRR) configures the baud rate for the specific USART port configured.



30.6.8 USART register map

The table below gives the USART register map and reset values.

Table 149. USART register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	USART_SR	Reserved																						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
	Reset value																							0	0	1	1	0	0	0	0	0	0
0x04	USART_DR	Reserved																						DR[8:0]									
	Reset value																							0	0	0	0	0	0	0	0	0	0
0x08	USART_BRR	Reserved																DIV_Mantissa[15:4]												DIV_Fraction [3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	USART_CR1	Reserved															OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
	Reset value																0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	USART_CR2	Reserved																LINEN	STOP [1:0]		CLKEN	CPOL	CPHA	LBCL	Reserved	LBDIE	LBDL	Reserved	ADD[3:0]			
	Reset value																	0	0	0	0	0	0	0		0	0		0	0	0	0	0
0x14	USART_CR3	Reserved																						ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
	Reset value																							0	0	0	0	0	0	0	0	0	0	0	0
0x18	USART_GTPR	Reserved																GT[7:0]										PSC[7:0]					
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Refer to *Section 2.3: Memory map* for the register boundary addresses.

Fig. 4. USART Peripheral Register Map

Additionally, the conversion between the USART register map shown in Fig. 4 and the implemented software drivers can be shown in Fig. 5 below, where 32-bit data represents the USART peripheral in a state of transmission and reception at a given time throughout the runtime of the project. Fig. 6 shows a user-friendly interface between the USART peripheral and the means by which data is exchanged between USART peripherals.

```c
// USART Peripheral Configuration Structure Definition
typedef struct
{
    uint8_t USART_Mode;
    uint32_t USART_Baud;
    uint8_t USART_StopBitNum;
    uint8_t USART_WordLength;
    uint8_t USART_ParityControl;
    uint8_t USART_HWFlowControl;
} USART_Config_t;

// USART Peripheral Handling Structure Definition
typedef struct
{
    USART_RegDef_t *pUSARTx;
    USART_Config_t USART_Config;
    uint8_t *pTXBuffer;
    uint8_t *pRXBuffer;
    uint32_t TXLength;
    uint32_t RXLength;
    uint8_t TXBusyState;
    uint8_t RXBusyState;
} USART_Handle_t;
```

Fig. 5. USART Peripheral Structure Definitions

```c
/* ********************************************** */
/* ******** API's supported by this driver ******** */
/* ********************************************** */

// Initialization Functions
void USART_Init(USART_Handle_t *pUSARTHandle);
void USART_DeInit(USART_RegDef_t *pUSARTx);

// Peripheral Clock Control
void USART_PeriClockControl(USART_RegDef_t *pUSARTx, uint8_t state);

// Data IO
void USART_SendData(USART_Handle_t *pUSARTx, uint8_t *pTXBuffer, uint32_t length);
void USART_ReceiveData(USART_Handle_t *pUSARTx, uint8_t *pRXBuffer, uint32_t length);
uint8_t USART_SendDataIT(USART_Handle_t *pUSARTHandle, uint8_t *pTXBuffer, uint32_t length);
uint8_t USART_ReceiveDataIT(USART_Handle_t *pUSARTHandle, uint8_t *pRXBuffer, uint32_t length);

// IRQ Configuration and ISR Handling
void USART_IRQInterruptConfig(uint8_t IRQNum, uint8_t state);
void USART_IRQPriorityConfig(uint8_t IRQNum, uint8_t IRQPriority);
void USART_IRQHandling(USART_Handle_t *pUSARTHandle);

// Other Peripheral Control APIs
void USART_SetBaudRate(USART_RegDef_t *pUSARTx, uint32_t BaudRate);
void USART_PeripheralControl(USART_RegDef_t *pUSARTx, uint8_t state);
uint8_t USART_GetFlagStatus(USART_RegDef_t *pUSARTx, uint32_t statusFlagName);
void USART_ClearFlag(USART_RegDef_t *pUSARTx, uint16_t statusFlagName);
void USART_ApplicationEventCallback(USART_Handle_t *pUSARTHandle, uint8_t event);
```

Fig. 6. USART API Definintions

Implemented API's such as "USART_IRQInterruptConfig()" in Fig. 6 can configure USART peripherals as interrupts, where the main loop halts running code in order to intercept new transmitted or received messages passing through the peripheral.

II. Methodology

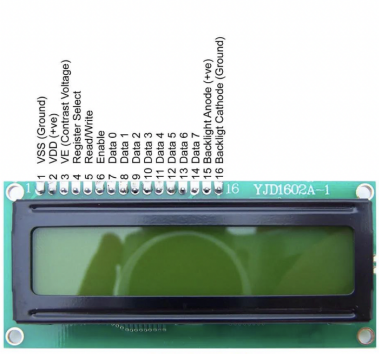
The following sections will display key implementation notes following schematic information of the IR transceiver implementation.

A. LCD Interface

A liquid crystal display (LCD) will be used as an output GPIO peripheral for receiving data via the USART protocol. Both GPIO inputs and outputs characterize the LCD; outputs from the MCU to the LCD will command the initiation and cursor positions of the display along with character data to be displayed in the form of 8-bit character commands.

Observing Fig. 7, the pinout of the LCD encompasses the necessary power terminals along with the data terminals used for communication. It is important to note that the LCD contains two main registers – the command and data register. Toggling the register select (RS) pin of the LCD with a GPIO pulse switches between the command and data registers, where command registers configures the LCD (initialization, cursor position, etc.) and the data register displays 8-bit characters. The read/write terminal of the LCD configures whether information will be written to or read from the display. For purposes specific to the IR transceiver implementation, the read/write terminal will remain constant to only write data to the LCD. The

LCD's enable pin locks in a specific instruction, whether the instruction consists of 8-bit data or 8-bit command to configure the display.



LCD pin definitions

PIN no.	Name
1	GND
2	VCC(+5V)
3	VEE
4	RS
5	R/W
6	E
7	D0
8	D1
9	D2
10	D3
11	D4
12	D5
13	D6
14	D7
15	LED+
16	LED-

Fig. 7. LCD IO Pinout Schematic

Observing Fig. 8, a list of LCD commands to configure the display are described. For this implementation, the algorithm necessary for initializing the LCD's data display is as follows –

- 0x30 — LCD initialization
- 0x38 — LCD mode select: 8-bit data
- 0x01 — Clear LCD display
- 0x02 — Cursor set to "Home" position
- 0x0F — Display is "On" and the cursor is blinking

Code (Hex)	Command to LCD Command Register
0x1	Clear Display Screen
0x2	Return Cursor Home
0x6	Increment Cursor (shift to the right)
0xF	Display On, Cursor Blinking
0x80	Force Cursor to Beginning of First Line
0xC0	Force Cursor to Beginning of Second Line
0x38	Mode: 2 lines and 5x7 characters (8-bit Data)
0x28	Mode: 2 lines and 5x7 characters (4-bit Data)
0x30	Initialization Command

Fig. 8. LCD Register Commands

Commands are converted from 8-bit hexadecimal values to 8-bit binary values such that the GPIO D0-7 pins can configure the LCD commands and data explicitly. Commands and character data are transmitted via GPIO pins D0-7, the only difference between which is how the RS terminal is preset, and if character data is being transmitted, the cursor must be shifted to the right in place for a new character. The software for transmitting LCD data explicitly can be found in Fig. 19 in the Appendix.

B. 4x4 Keypad Interface

A 4x4 keypad will be used as an input GPIO peripheral for transmitting data via the USART protocol. Fig. 9 shows that the 16-input interface is cross-hatched through only 8 GPIO pins. The press of a key in conjunction with its corresponding row set to a LOW state will also set the corresponding column to a LOW state. Specifically, if the MCU writes a row to a LOW state while simultaneously reading a column as a LOW state, then the software will save the corresponding key pressed. The scanning feature of the keypad in implemented in Fig. 20 in the Appendix.

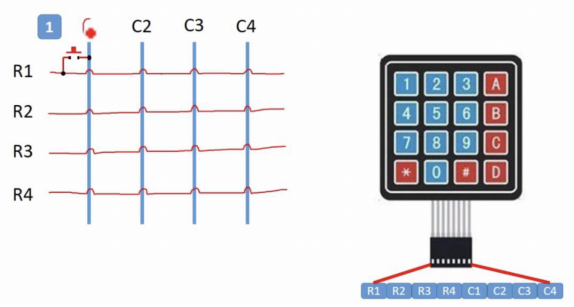


Fig. 9. 4x4 Keypad IO Pinout Schematic

Something to note is that while the keypad interface is cycling through its scan function (Fig. 20), internal pull-up resistors are enabled for each column pin, thus ensuring that there are no floating point errors associated with a key press. Referring back to Fig. 2, the column pins are enabled as inputs with pull-up internal resistors. Fig. 10 shows that while row pins can output a LOW state, a key press can move the corresponding column to a LOW state as well, thus bypassing the internal pull-up resistor and disabling the input pin. The disabled input pin is the functional condition for marking an input key.

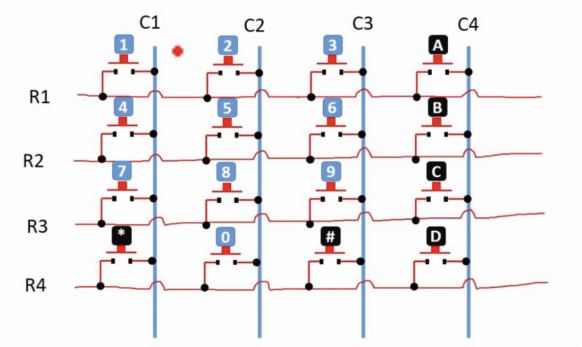


Fig. 10. 4x4 Keypad Pull-up Switching Schematic

C. IR TX/RX Circuit

Fig. 11 below depicts the schematic for the implemented IR transceiver with its respective receiving and transmitting sub-circuits. The IR transmitter transmits 38 kHz pulses with the help of a NE555P timer, while a VS1838 IR receiver accepts the pulses and inverts the signal with the help of a CD4069 hex inverter. The transmitted and received signal are connected to a four-pin terminal that will bus to the MCU's USART peripheral.

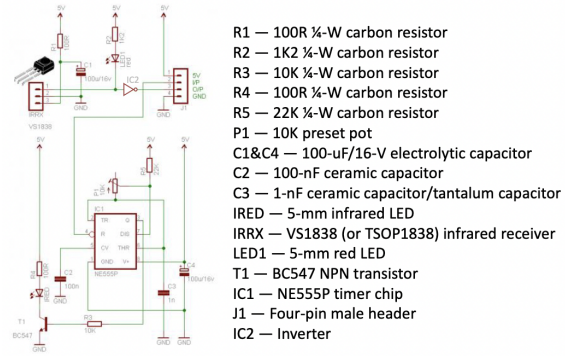


Fig. 11. IR Transceiver Schematic

III. Results

The following section documents the completed prototype of the IR transceiver system.

A. System Build

Fig. 12, 13, 14, 15 below show the completed prototype of the IR transceiver implementation, where the keypad interface inputs characters processed by the MCU and is transmitted to a receiver via an IR LED. The receiver then processes the transmitted data through the MCU and then decodes the keypad's character data on the LCD interface.

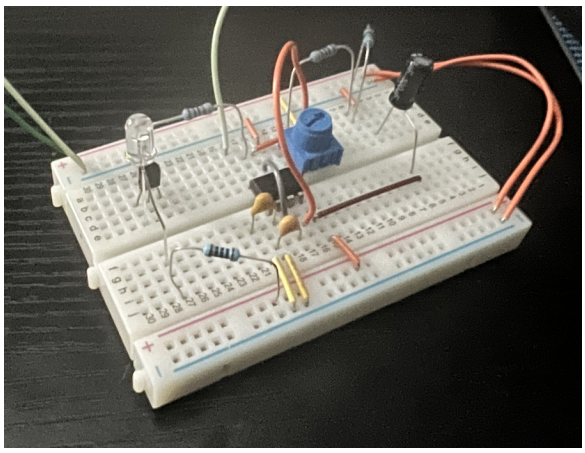


Fig. 12. IR Transmitter Prototype

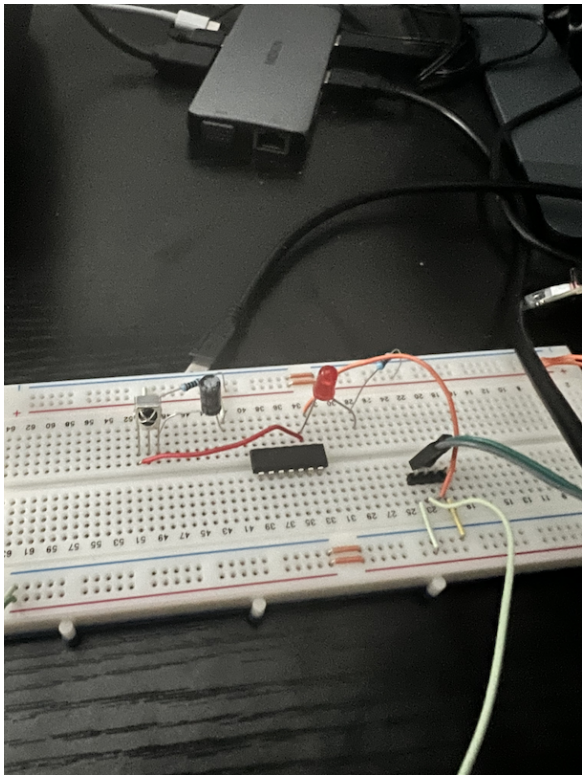


Fig. 13. IR Receiver Prototype

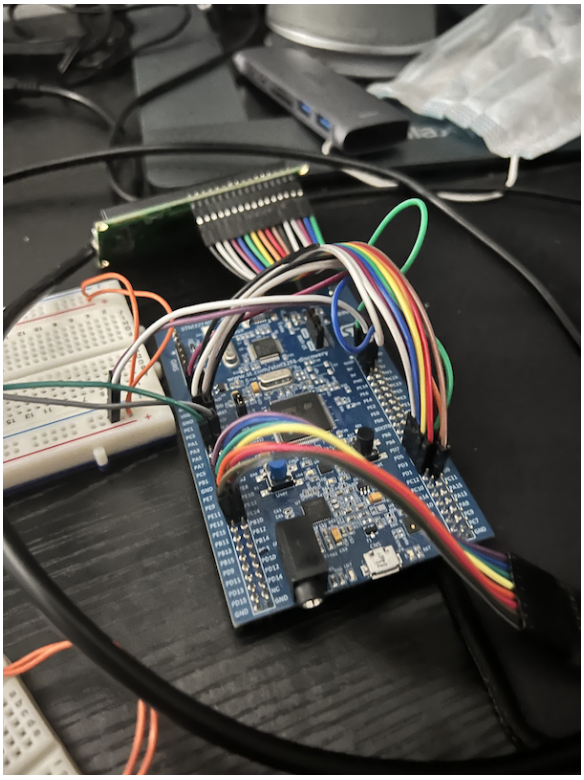


Fig. 14. MCU Input and Output Configuration

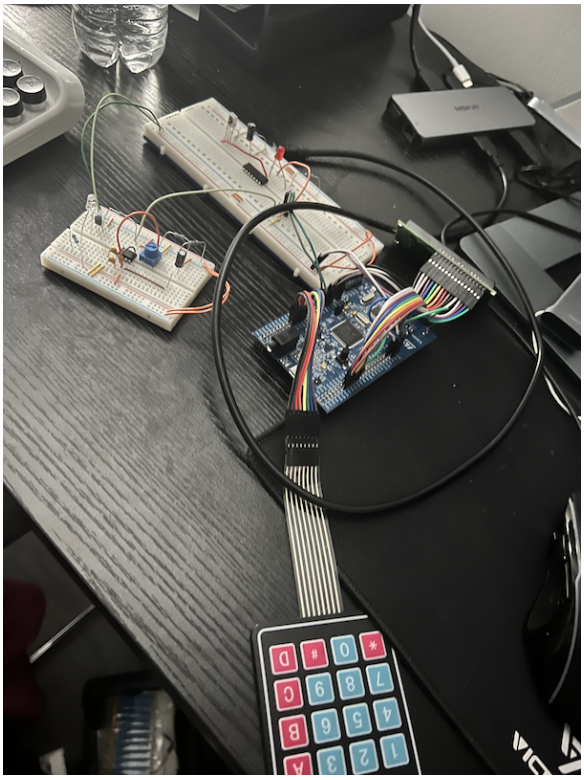


Fig. 15. IR Transceiver System Prototype

B. USART TX/RX Loop

Fig. 16 below shows how the USART TX/RX subcomponents interact for as long as the program is active, where USART2 is initialized to transmit messages and USART3 is initialized to receive messages. Both USART2 and USART2 are configured with interrupts through GPIOA's alternate function #7.

```
int main(void)
{
    Keypad_Init();
    LCD_Init();

    USART_GPIO_INIT();
    USART_TX_INIT();
    USART_RX_INIT();

    uint8_t msg_tx[1], msg_rx[1];

    while(1)
    {
        // TX ROUTINE: Send keypad data to TX pin
        msg_tx[0] = Keypad_Scan();
        while(USART_SendDataIT(&usart_handle_tx, (uint8_t*)msg_tx, 1) != USART_READY);
        USART_SendData(&usart_handle_tx, (uint8_t*)msg_tx, 1);

        // RX ROUTINE: Send RX data to LCD
        while(USART_ReceiveDataIT(&usart_handle_rx, (uint8_t*)msg_rx, 1) != USART_READY);
        USART_ReceiveData(&usart_handle_rx, (uint8_t*)msg_rx, 1);
        LCD_Data((uint8_t*)msg_rx);
    }

    return 0;
}
```

Fig. 16. IR Transceiver Main Loop

While the system is not receiving any messages, the program checks for transmissions by scanning the GPIO ports connected to the input keypad interface. Concurrently, while the system is not transmitting any messages, the program checks for received data then to be displayed on the LCD.

IV. Conclusion

Altogether, a knowledge of the STM32F407's register configurations allowed for an intuitive implementation of a simple IR transceiver. Further development of this implementation may include the transfer from a breadboard prototype to a PCB, along with experimenting with different frequencies of the receiver.

V. APPENDIX

```c
void GPIO_WriteToOutputPin(GPIO_RegDef_t *pGPIOx, uint8_t pinNum, uint8_t data)
{
    if(data == GPIO_PIN_SET) {pGPIOx -> ODR |= (1 << pinNum);}
    else {pGPIOx -> ODR &= ~(1 << pinNum);}
}
```

Fig. 17. GPIO Output Write Function

```c
uint8_t GPIO_ReadFromInputPin(GPIO_RegDef_t *pGPIOx, uint8_t pinNum)
{
    uint8_t value;
    value = (uint8_t)((pGPIOx -> IDR >> pinNum) & 0x00000001);
    return value;
}
```

Fig. 18. GPIO Input Read Function

```c
void LCD_Data(unsigned char data)
{
   // Set RS to HIGH
   GPIO_WriteToOutputPin(GPIOC, GPIO_PIN_NUM_0, ENABLE);

   // Put data on GPIO pins (D0 - D7)
   if(((uint8_t)data & LCD_PD7_MASK) == LCD_PD7_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_7, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_7, DISABLE);}

   if(((uint8_t)data & LCD_PD6_MASK) == LCD_PD6_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_6, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_6, DISABLE);}

   if(((uint8_t)data & LCD_PD5_MASK) == LCD_PD5_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_5, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_5, DISABLE);}

   if(((uint8_t)data & LCD_PD4_MASK) == LCD_PD4_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_4, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_4, DISABLE);}

   if(((uint8_t)data & LCD_PD3_MASK) == LCD_PD3_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_3, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_3, DISABLE);}

   if(((uint8_t)data & LCD_PD2_MASK) == LCD_PD2_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_2, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_2, DISABLE);}

   if(((uint8_t)data & LCD_PD1_MASK) == LCD_PD1_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_1, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_1, DISABLE);}

   if(((uint8_t)data & LCD_PD0_MASK) == LCD_PD0_MASK) {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_0, ENABLE);}
   else                                               {GPIO_WriteToOutputPin(GPIOD, GPIO_PIN_NUM_0, DISABLE);}
   LCD_Enable();
   LCD_Command(0x06); // Shift cursor to the right for the next character
}
```

Fig. 19. LCD Data Transmission Function

```c
    // check row 0 of keypad by setting it to LOW
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_7, DISABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_8, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_9, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_10, ENABLE);
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_11) == DISABLE) {key = '1';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_12) == DISABLE) {key = '2';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_13) == DISABLE) {key = '3';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_14) == DISABLE) {key = 'A';}

    // check row 1 of keypad by setting it to LOW
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_7, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_8, DISABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_9, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_10, ENABLE);
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_11) == DISABLE) {key = '4';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_12) == DISABLE) {key = '5';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_13) == DISABLE) {key = '6';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_14) == DISABLE) {key = 'B';}

    // check row 2 of keypad by setting it to LOW
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_7, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_8, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_9, DISABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_10, ENABLE);
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_11) == DISABLE) {key = '7';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_12) == DISABLE) {key = '8';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_13) == DISABLE) {key = '9';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_14) == DISABLE) {key = 'C';}

    // check row 3 of keypad by setting it to LOW
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_7, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_8, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_9, ENABLE);
    GPIO_WriteToOutputPin(GPIOE, GPIO_PIN_NUM_10, DISABLE);
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_11) == DISABLE) {key = '*';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_12) == DISABLE) {key = '0';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_13) == DISABLE) {key = '#';}
    if(GPIO_ReadFromInputPin(GPIOE, GPIO_PIN_NUM_14) == DISABLE) {key = 'D';}

    Keypad_Delay(10);

    return key;
}
```

Fig. 20. 4x4 Keypad Input Scanning Function