A Dual-Node "Walkie-Talkie" over USART on AVR
Channelized Link with On-Link Encryption, LCD UI, and Periodic Standby

Parsa Haghighatgoo - 40030644

September 16, 2025

Abstract

This report documents the design and implementation of a two-node digital "walkie-talkie" built on AVR microcontrollers using USART. Each node can act as a transmitter or receiver, supports multiple user-selectable channels (implemented as baud-rate bands mapped from a potentiometer), provides an LCD user interface and status LEDs, debounced input buttons for canned messages, and a lightweight link-layer encryption (XOR per channel). A periodic *Standby* heartbeat verifies mutual coverage. The full source and the Proteus schematic are included. *Source and original brief/spec provided by the project files.*

1 Project Overview

The system consists of two identical AVR boards configured as **Master** and **Slave** in Proteus (either can run in Tx/Rx mode). Communication is full digital over UART with a simple framing: a sync byte followed by an encrypted one-byte payload (mapped to text on the receiver's LCD).

A summary of the functional goals (from the assignment brief) includes:

- Channel selection via a potentiometer, mapping to discrete baud rates.
- LCD to display received status text; push buttons to send canned codes.
- A sync byte check (green LED on success, red LED on mismatch).
- Periodic *Standby* message as a coverage heartbeat.
- Optional link encryption and support for additional channels/messages.

2 Hardware Architecture

2.1 Block Diagram & Major Components

Each node contains:

- AVR MCU (8 MHz internal/external clock).
- LCD (8-bit parallel: data on `PORTC`, control on `PA0-PA2`).
- Potentiometer on ADC channel (PA3) for channel selection.
- Three push buttons for message triggers; one Mode button (Tx/Rx).
- Two LEDs (common-anode) on `PB3` (red) and `PB4` (green).

2.2 Proteus Schematic

Figures 1, 2, and 3 show the full design and per-node wiring from the Proteus project.

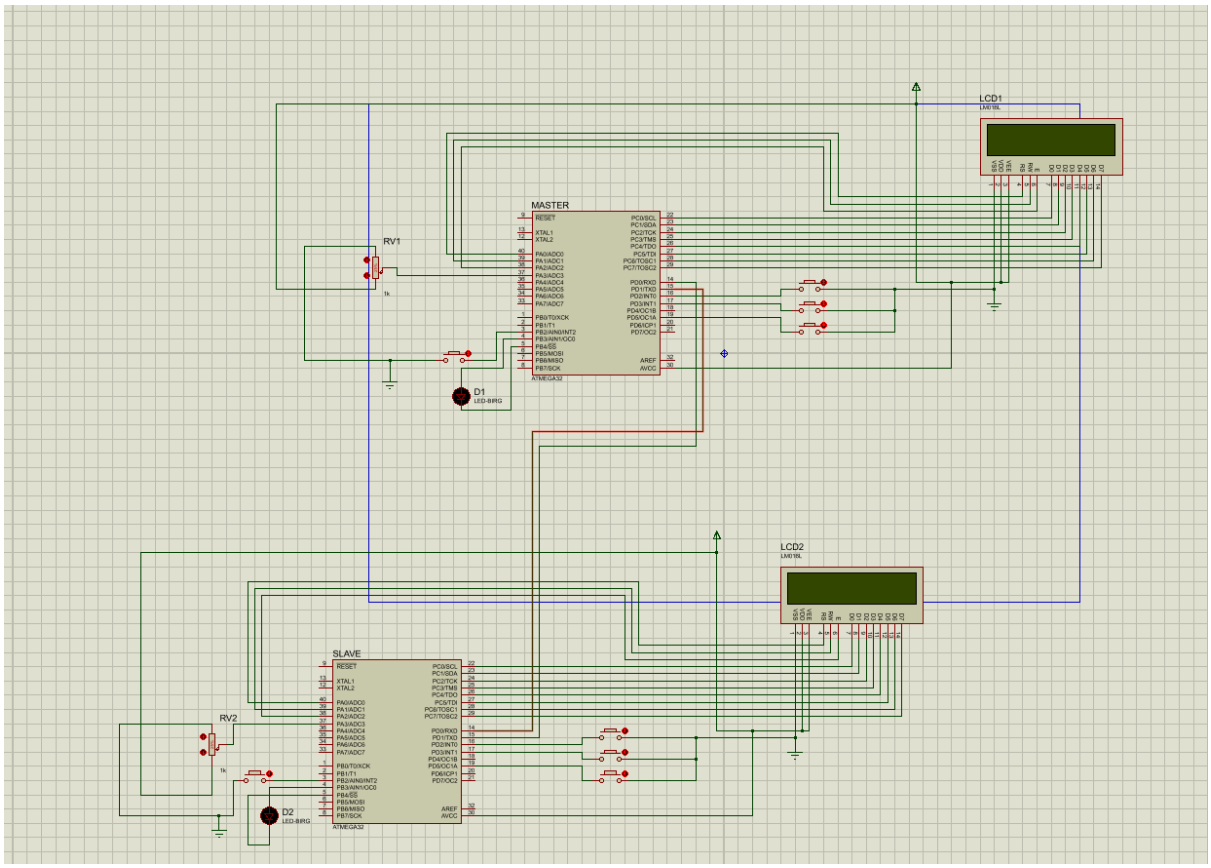


Figure 1: Full Proteus implementation (two nodes with LCDs and channel bus).

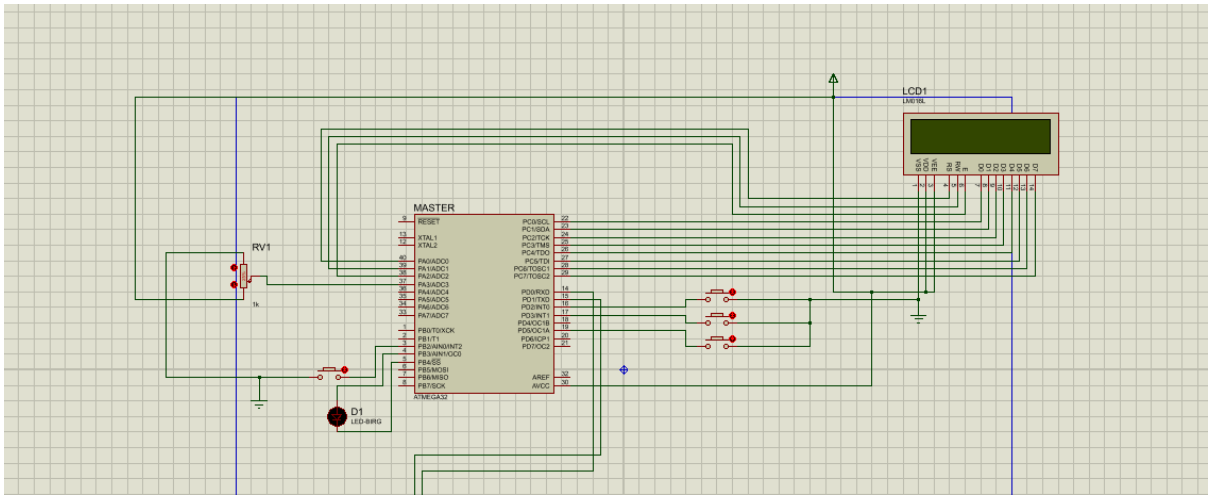


Figure 2: Master node wiring (LCD on PORTC/PORTA, LEDs on PB3/PB4, buttons, ADC).

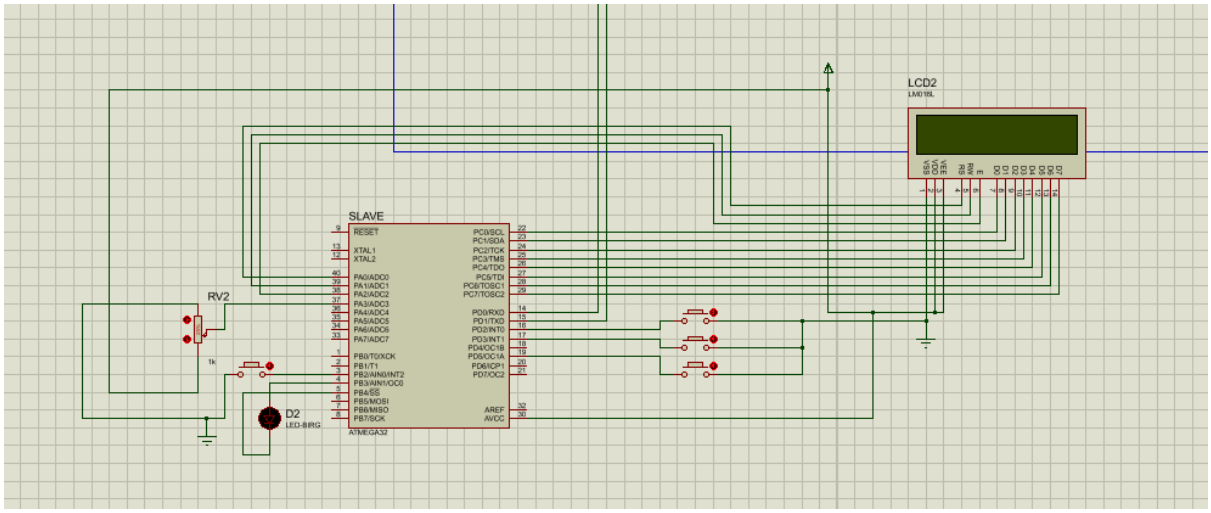


Figure 3: Slave node wiring; identical interface to Master for symmetry.

3 Communication Design

3.1 Frame Format

Each transmission is two bytes:

1. **SYNC** (`0xAA`) — sent in clear to align the receiver.

2. **Payload** — one ASCII letter, XOR-encrypted with the channel's per-key value.

At the receiver, a correct SYNC lights the green LED and decrypts the payload; mismatch lights the red LED. The mapping from payload letters to phrases is:

Code	Displayed Text
`H`	Help :(
`O`	Over :)
`R`	Roger That :D
`S`	Stand By XD
`A`	ACK :)
`B`	BUSY :|
`Y`	YES!

3.2 Channels, Baud Rates, and Keys

A potentiometer on `ADC3` maps [0..1023] to 8 discrete channels ($v \gg 7 \Rightarrow 0..7$). Each channel sets a baud rate and a simple XOR key used for payload encryption/decryption. The implemented table is:

CH	Baud	XOR Key
0	4800	0x11
1	9600	0x22
2	19200	0x33
3	38400	0x44
4	57600	0x55
5	76800	0x66
6	115200	0x77
7	250000	0x88

Note: CH7 at 250 kbaud is fine in simulation; on real hardware prefer 16 MHz or select 57600/115200 as appropriate. Channel and key logic, SYNC and message handling are all implemented in the provided code.[1]

3.3 Heartbeat (*Standby*)

Timer1 runs in CTC mode with a 1 Hz compare (`OCR1A=7812` @ 8 MHz with 1024 prescale). After ∼3 ticks the transmitter sends `S` (Stand By) automatically to verify coverage while idle. This threshold is easy to tune in firmware.

[1] See `CHANNELS[]` table, SYNC `0xAA`, and `enc()/dec()` helpers in the source.

4 Firmware Architecture

4.1 Key Modules

- **LCD** (8-bit mode): `init_LCD()`, `LCD_cmd()`, `LCD_write()`.
- **LED control**: common-anode green/red helpers.
- **ADC**: AVcc reference, prescaler 64, channel select with masking.
- **USART**: 8N1, interrupt-capable; `USART_Init()` derives `UBRR` from selected channel's baud.
- **Timer1**: 1 Hz tick for heartbeat timing.
- **Buttons**: debounced with short/long-press detection ($\geq$600 ms = long). Buttons map to {H/A}, {O/B}, {R/Y}.
- **Sleep**: `SLEEP_MODE_IDLE` between events to reduce power.

4.2 Operating Modes

A Mode button toggles between **Tx** and **Rx**. LCD briefly shows `CH# Tx` or `CH# Rx` after a change. In Tx, short/long presses immediately send the mapped encrypted code. In Rx, incoming bytes are processed only after a correct SYNC.

5 Testing and Results

We validated:

1. Channel separation: nodes must be on the same channel (baud + key) to decode correctly; otherwise the SYNC check fails (red LED).
2. Message mapping: all seven codes render correctly on LCD.
3. Heartbeat: `S` is emitted after an idle period ($\sim$3 s by default).
4. Sleep and wake behavior: nodes idle in `IDLE` mode and wake on button/RX/compare events.

6 Bill of Materials (per node)

- 1$\times$ AVR MCU (e.g., ATmega16/32, 8 MHz)
- 1$\times$ 16$\times$2 Character LCD (HD44780 compatible)
- 1$\times$ Potentiometer (10 kΩ) for channel select
- 3$\times$ Push buttons (message), 1$\times$ Push button (mode)
- 2$\times$ LEDs (common anode) + resistors
- Misc: headers, wires; optional external crystal, decoupling caps

7 LCD Subsystem (HD44780, 8-bit)

7.1 Purpose & Requirements Mapping

This module drives a 16×2 HD44780-compatible character LCD using an 8-bit parallel interface: data bus on `PORTC` and control pins `RS=PA0`, `RW=PA1`, `EN=PA2`. It provides routines to initialize the display, send commands, and write characters. This directly implements the *"LCD to display received status/messages"* requirement from the project description (LCD) and enables the UI feedback described alongside the message buttons.[2]

7.2 How It Works (Timing & Signals)

The interface follows the HD44780 timing:

- Write the full 8-bit command/data to `PORTC`.
- Set `RS=0` for commands or `RS=1` for data; keep `RW=0` (write-only).
- Generate a short `EN` high-pulse to latch the byte, then pull it low.

Initialization sends (with conservative delays instead of reading the busy flag): `0x38` (8-bit, 2-line), `0x01` (clear), `0x0E` (display on, cursor on), and `0x80` (DDRAM address 0). These calls are spaced by 1–2 ms delays, which exceed the LCD's worst-case command/clear timing and are safe at 8 MHz.

Gotcha (PORT writes). The control helper writes the entire `PORTA` at once. Ensure only `PA0..PA2` are used for LCD control or mask other bits if needed.

7.3 Verification

In Proteus (and on hardware), after reset the LCD should clear, enable the cursor, and place the cursor at home. Sending a character via `LCD_write` should display it immediately; e.g., writing "CH0" after channel selection.

7.4 Code

Listing 1: LCD driver (8-bit mode on PORTC/PORTA)

```
#define LCD_DATA PORTC
#define ctrl PORTA
#define en PA2
#define rw PA1
#define rs PA0

void LCD_cmd(unsigned char cmd);
void LCD_write(unsigned char data);

void init_LCD(void)
{
    LCD_cmd(0x38);       // 8-bit, 2 lines, 5x8
    _delay_ms(1);
    LCD_cmd(0x01);       // clear
    _delay_ms(1);
    LCD_cmd(0x0E);       // display on, cursor on
    _delay_ms(1);
```

[2] See the project brief sections and the message table that require showing the decoded phrase on the LCD.

```
18      LCD_cmd(0x80);       // DDRAM 0 (home)
19      _delay_ms(1);
20  }
21
22  void LCD_cmd(unsigned char cmd)
23  {
24      LCD_DATA = cmd;
25      ctrl = (0<<rs) | (0<<rw) | (1<<en); // RS=0, RW=0, EN=1
26      _delay_ms(1);
27      ctrl = 0x00;                        // EN=0
28      _delay_ms(2);
29  }
30
31  void LCD_write(unsigned char data)
32  {
33      LCD_DATA = data;
34      ctrl = (1<<rs) | (0<<rw) | (1<<en); // RS=1, RW=0, EN=1
35      _delay_ms(1);
36      ctrl = (1<<rs) | (0<<rw) | (0<<en); // EN=0
37      _delay_ms(2);
38  }
```

8 Message Definitions and Mapping

8.1 Purpose & Requirements Mapping

This section defines the symbolic messages exchanged between nodes. According to the project description, each button press must generate a one-character code that is transmitted over the channel, and the receiver must expand that code into a meaningful phrase on its LCD. The requirement in the spec is handled here. Additionally, the spec requires that every transmission begin with a fixed `SYNC` byte to verify correct channel alignment — implemented with `SYNC_CODE = 0xAA`.

8.2 How It Works

- `SYNC_CODE` is transmitted first, in clear, for every packet. If the receiver sees `0xAA`, it lights the green LED and proceeds; otherwise the red LED indicates a mismatch.
- Message pointers map single-letter codes (`'H','O','R','S','A','B','Y'`) to longer strings shown on the LCD. This matches the brief's requirement for "Help", "Over", "Roger That", etc.
- The "Stand By" message (`'S'`) is also included here, used by the heartbeat timer to automatically send periodic coverage checks.

8.3 Verification

In Proteus, pressing each button must send its code. The receiver's LCD should display the full text string (e.g., "HELP :(" when `'H'` is sent). If the SYNC byte is corrupted or channel mismatch occurs, only the red LED lights and no message is shown.

8.4 Code

Listing 2: Message definitions and sync code

```c
#define SYNC_CODE 0xAA

const char *msg_H = "HELP :(";
const char *msg_O = "OVER :)";
const char *msg_R = "Roger That :D";
const char *msg_S = "Stand By XD";
const char *msg_A = "ACK :)";
const char *msg_B = "BUSY :|";
const char *msg_Y = "YES!";
```

9 LED Control Subsystem

9.1 Purpose & Requirements Mapping

This module drives two LEDs (red and green) connected to `PB3` and `PB4` on the AVR. The LEDs are wired in a common-anode configuration, so logic `LOW` turns the LED on, and logic `HIGH` turns it off. According to the project description, the system must indicate whether the synchronization byte and communication were successful:

- Green LED lights when the SYNC code matches and communication is valid.
- Red LED lights when the SYNC code is incorrect or the channel mismatch prevents decoding.

This directly implements the requirement: *"The device must have two LEDs. Green lights when communication is correct, red lights when communication fails."*

9.2 How It Works

- `LED_Init()` configures both pins as outputs.
- `LED_Off()` drives both pins high, turning both LEDs off.
- `LED_Red()` sets red pin low (on) and green pin high (off).
- `LED_Green()` sets green pin low (on) and red pin high (off).

Because the LEDs are common-anode, the logic is inverted compared to common-cathode wiring.

9.3 Verification

In simulation (Proteus) or hardware, a successful message reception with a valid SYNC byte should make the green LED blink briefly. If a wrong SYNC byte is received or baud/channel mismatch occurs, the red LED should blink instead.

9.4 Code

Listing 3: Red/Green LED control (common-anode)

```c
#define RED_PIN   PB3
#define GREEN_PIN PB4

static inline void LED_Init(void) {
    // Configure both pins as outputs once
    DDRB |= (1 << RED_PIN) | (1 << GREEN_PIN);
}
```

```
8
9  void LED_Off(void) {
10     LED_Init();
11     // Common-anode: HIGH turns LED off
12     PORTB |= (1 << RED_PIN) | (1 << GREEN_PIN);
13 }
14
15 void LED_Red(void) {
16     LED_Init();
17     PORTB &= ~(1 << RED_PIN);   // LOW = ON red
18     PORTB |=  (1 << GREEN_PIN); // HIGH = OFF green
19 }
20
21 void LED_Green(void) {
22     LED_Init();
23     PORTB &= ~(1 << GREEN_PIN); // LOW = ON green
24     PORTB |=  (1 << RED_PIN);   // HIGH = OFF red
25 }
```

10 ADC and Channel Selection

10.1 Purpose & Requirements Mapping

This section reads a potentiometer on ADC channel `PA3` and maps its percentage to discrete communication channels (baud rates). It implements the requirement from the project description: *"Use a potentiometer to select the channel; each channel has a different baud rate; if devices are on different channels they cannot read each other's messages."* The brief also specifies the four bands and baud mapping: 0–25%→4800, 26–50%→9600, 51–75%→19200, 76–100%→115200.

10.2 How It Works

ADC setup. `ADC_Init()` selects AVcc as reference (`REFS0`) and enables the ADC with prescaler 64, giving an ADC clock of 125 kHz at 8 MHz. `ADC_Read(ch)` masks the channel bits, updates `ADMUX`, starts a conversion, waits for `ADSC` to clear, and returns the 10-bit result. :contentReference[oaicite:3]index=3

Channel mapping. `select_channel()` converts the 10-bit reading to % (0..1023 ↦ 0..100) and chooses a baud from four thresholds that match the spec's bands. This keeps both nodes "tuned" the same when their pots are in the same region. :contentReference[oaicite:4]index=4

10.3 Verification

In Proteus, sweep the potentiometer and print the selected channel/baud on the LCD (e.g., "CH/BAUD"). Two devices should successfully exchange messages only when their pots sit in the same band; otherwise sync fails as designed (see Sec. 11). :contentReference[oaicite:5]index=5

10.4 Code

Listing 4: ADC setup and potentiometer-based channel selection

```
1 #define ADC_PRESCALER     ((1 << ADPS2) | (1 << ADPS1))   // Prescaler =
      64
2 #define ADC_REF_AVCC      (1 << REFS0)                    // Reference =
      AVcc
3 #define ADC_CHANNEL_MASK  0x07                            // Only lower 3
       bits valid
```

```
4  #define ADC_CLEAR_MASK    0xF8                                  // Clear
       channel bits
5
6  void ADC_Init(void) {
7      // Select AVcc as reference, start with channel 0
8      ADMUX  = ADC_REF_AVCC;
9      // Enable ADC with prescaler
10     ADCSRA = (1 << ADEN) | ADC_PRESCALER;
11 }
12
13 uint16_t ADC_Read(uint8_t channel) {
14     // Keep only the valid channel bits (0  7 )
15     channel &= ADC_CHANNEL_MASK;
16     // Set channel while preserving reference selection
17     ADMUX = (ADMUX & ADC_CLEAR_MASK) | channel;
18
19     // Start conversion
20     ADCSRA |= (1 << ADSC);
21     // Wait for conversion to finish
22     while (ADCSRA & (1 << ADSC));
23
24     return ADC; // Return 10-bit result
25 }
26
27 //-----------------------------------------------
28 // BAUD_RATE Selection
29
30 #define ADC_CHANNEL_POT   3
31 #define BAUD_25   4800UL
32 #define BAUD_50   9600UL
33 #define BAUD_75   19200UL
34 #define BAUD_100  115200UL
35
36 unsigned long select_channel(void) {
37     uint16_t pot_value = ADC_Read(ADC_CHANNEL_POT);   // Read
          potentiometer on PA3
38     uint8_t percentage = (uint32_t)pot_value * 100 / 1023;
39
40     unsigned long baud_rate;
41
42     if (percentage <= 25) {
43         baud_rate = BAUD_25;
44     } else if (percentage <= 50) {
45         baud_rate = BAUD_50;
46     } else if (percentage <= 75) {
47         baud_rate = BAUD_75;
48     } else {
49         baud_rate = BAUD_100;
50     }
51
52     return baud_rate;
53 }
```

11 USART Driver (8N1, Variable Baud)

11.1 Purpose & Requirements Mapping

This driver configures the AVR USART for 8 data bits, 1 stop bit, no parity (8N1), with a **variable baud rate** chosen by the channel-selection logic above. It satisfies the requirement that both transmitter and receiver compute their baud rate from the channel and communicate using USART. *"Sender and receiver determine their baud rate; the sender sends a sync character first, then the message character; the receiver checks sync and then reads the next byte."* :contentReference[oaicite:7]index=7

11.2 How It Works

UBRR calculation. For asynchronous normal mode, the baud register is UBRR $= \frac{F_{CPU}}{16 \cdot \text{baud}} - 1$. With $F_{CPU} = 8\,\text{MHz}$, `USART_Init(baud)` computes UBRR, writes `UBRRH:UBRRL`, enables RX/TX and RX interrupt, and sets the frame format via `UCSRC` (using `URSEL` to target UCSRC). Data is written when `UDRE` is set and received when `RXC` is set. :contentReference[oaicite:8]index=8

11.3 Verification

- Put two nodes on the same channel (e.g., 9600). Transmit: the receiver should read SYNC then the payload without framing errors. - Change the pot on one node to another channel; the receiver should show a sync failure (your red LED section handles this). :contentReference[oaicite:9]index=9

11.4 Code

Listing 5: USART initialization and blocking TX/RX

```
#define USART_DATA_BITS   ((1 << UCSZ1) | (1 << UCSZ0)) // 8-bit data
#define USART_ENABLE      ((1 << RXEN) | (1 << TXEN) | (1 << RXCIE))
#define USART_URSEL_MASK  (1 << URSEL)  // Needed to write UCSRC

void USART_Init(unsigned long baud) {
    uint16_t ubrr_value = (F_CPU / (16UL * baud)) - 1;

    // Set baud rate
    UBRRH = (uint8_t)(ubrr_value >> 8);
    UBRRL = (uint8_t)(ubrr_value);

    // Enable RX, TX and RX interrupt
    UCSRB = USART_ENABLE;

    // Set frame: 8 data bits, 1 stop bit, no parity
    UCSRC = USART_URSEL_MASK | USART_DATA_BITS;
}

void USART_Transmit(uint8_t data) {
    // Wait until buffer is empty
    while (!(UCSRA & (1 << UDRE)));
    UDR = data;
}

uint8_t USART_Receive(void) {
    // Wait until data is received
    while (!(UCSRA & (1 << RXC)));
```

```
28      return UDR;
29  }
```

12 Power Management and Heartbeat (Stand-By)

12.1 Purpose & Requirements Mapping

This section implements two behaviors required by the project description:

1. **Sleep/idle after activity**: the receiver should go to sleep after processing a message; choice of sleep mode is up to the designer (we use `IDLE`).

2. **Periodic Stand-By message**: the transmitter should automatically send a *Stand By* message (recommended code: `'S'`) after a fixed interval to verify both devices are in coverage and on the same channel.

These exactly match the brief's "sleep after receive" and "send periodic Stand-By to verify link" requirements. :contentReference[oaicite:3]index=3 :contentReference[oaicite:4]index=4

12.2 How It Works

Sleep. `go_to_sleep()` enables sleep, sets `SLEEP_MODE_IDLE`, executes `sleep_cpu()`, then disables sleep to prevent accidental re-entry. The main loop calls it after receiving or sending, so the MCU idles until the next event (button, UART RX, or timer). :contentReference[oaicite:5]index=5

Timer1 Heartbeat. `Timer1_Init()` puts Timer1 in CTC mode with OCR1A as the top, prescaler 1024, and OCR1A set for a 1 Hz interrupt at 8 MHz:

$$\text{OCR1A} = \frac{F_{\text{CPU}}}{1024 \cdot 1} - 1 = 7812.$$

The ISR increments `standby_counter` and, when the threshold is reached, sets `send_standby_flag` for the main loop to transmit `'S'`. *Note:* With a 1 Hz tick, `standby_counter >= 3` means ≈ 3 seconds, not "1 minute". Set the threshold to ≥ 60 for ≈ 60 seconds to match the brief. :contentReference[oaicite:6]index=6 :contentReference[oaicite:7]index=7

12.3 Verification

- In Tx mode, remain idle; after the threshold, the device should automatically send `'S'` and the peer's LCD should show the "Stand By" phrase (Section 8). On channel mismatch, the peer should fail SYNC and indicate red (Section 9).

- After a manual send/receive, observe the MCU dropping into IDLE and waking on the next IRQ (button, UART RX, or Timer1 compare).

12.4 Code

Listing 6: Sleep (IDLE) and Timer1 heartbeat at 1 Hz

```
1  static inline void go_to_sleep(void) {
2      // Configure sleep mode
3      sleep_enable();
4      set_sleep_mode(SLEEP_MODE_IDLE);
5
```

```
6       // Enter sleep
7       sleep_cpu();
8
9       // Wake-up: disable sleep to avoid accidental re-entry
10      sleep_disable();
11  }
12
13  //----------------------------------------
14  // Timer/Counter
15  volatile uint8_t standby_counter   = 0;
16  volatile uint8_t send_standby_flag = 0;
17
18  // 1 Hz compare value for F_CPU @ prescaler 1024
19  #define T1_PRESCALER_BITS   ((1 << CS12) | (1 << CS10))      // 1024
20  #define T1_CLEAR_PRESCALER  ((1 << CS12) | (1 << CS11) | (1 << CS10))
21  #define T1_CTC_MODE         (1 << WGM12)
22  #define T1_OCR1A_1HZ        ((uint16_t)((F_CPU / 1024UL) / 1UL) - 1U)
        // 8000000/1024 - 1 = 7812
23
24  ISR(TIMER1_COMPA_vect) {
25      standby_counter++;
26      if (standby_counter >= 3) {    // NOTE: ~3 s at 1 Hz; use >=60 for
            ~1 min per brief
27          send_standby_flag = 1;     // Set flag for main loop
28      }
29  }
30
31  void Timer1_Init(void) {
32      // CTC mode (OCR1A top)
33      TCCR1B &= ~T1_CLEAR_PRESCALER; // stop timer while configuring
34      TCCR1B |= T1_CTC_MODE;
35
36      // 1 Hz tick with F_CPU/1024 prescale
37      OCR1A = T1_OCR1A_1HZ;          // 7812 for 8 MHz
38
39      // Enable Compare Match A interrupt
40      TIMSK |= (1 << OCIE1A);
41
42      // Start timer with prescaler = 1024
43      TCCR1B |= T1_PRESCALER_BITS;
44  }
45
46  //----------------------------------------
47  // Empty ISR stubs (external IRQs + USART RX)
48  ISR(INT0_vect) {}
49  ISR(INT1_vect) {}
50  ISR(INT2_vect) {}
51  ISR(USART_RXC_vect) {}
```

13 Channel Table with Per-Channel Encryption & Auto-Apply

13.1 Purpose & Requirements Mapping

To support more channels and improve confidentiality (bonus), we define a table of channels, each with a baud rate and a simple XOR key. The currently selected channel's *baud* and *key* are applied automatically as the user turns the potentiometer. This implements:

- **Channelization**: different baud per channel; devices must be on the same channel to communicate.
- **Bonus: encryption**: encrypt before sending, decrypt at the receiver.

Both appear in the project description as requirements and bonus items. :contentReference[oaicite:10]index=10 :contentReference[oaicite:11]index=11

13.2 How It Works

- **Channel table** `CHANNELS[]` lists 8 channels with their baud and per-channel XOR key.
- **Current key** `ENC_KEY_CURRENT` is updated whenever the channel changes; `enc()/dec()` XOR the payload.
- **Indexing** `channel_index_from_adc()` maps ADC reading 0..1023 to 0..7 by dividing by 128.
- **Apply** `Channel_SelectAndApply(show)` re-reads the pot, updates global channel, sets the new key, re-initializes USART with the channel's baud, and optionally shows `CH#` on the LCD.

Note on baud choices. CH7 at 250 kbaud is fine in simulation; for real hardware, use a 16 MHz clock or pick a conservative baud (e.g., 57600/115200). :contentReference[oaicite:12]index=12

13.3 Verification

Rotate the potentiometer and confirm the LCD updates to `CH#` when the index changes. On the receiver, messages should only decode when both nodes are on the same channel; otherwise the SYNC check fails and red LED indicates error (Section 9).

13.4 Code

Listing 7: Channel table with per-channel XOR key and auto-apply

```
typedef struct {
    unsigned long baud;
    uint8_t       key;
} Channel;

#define NUM_CHANNELS 8
static const Channel CHANNELS[NUM_CHANNELS] = {
    { 4800UL,   0x11 },  // CH0
    { 9600UL,   0x22 },  // CH1
    { 19200UL,  0x33 },  // CH2
    { 38400UL,  0x44 },  // CH3
    { 57600UL,  0x55 },  // CH4
    { 76800UL,  0x66 },  // CH5 (ATmega accepts)
    { 115200UL, 0x77 },  // CH6
    { 250000UL, 0x88 },  // CH7 (sim OK; real HW prefer 16MHz or lower
        baud)
};

// Global "current" key selected by channel
static uint8_t ENC_KEY_CURRENT = 0x00;
static inline uint8_t enc(uint8_t b){ return b ^ ENC_KEY_CURRENT; }
static inline uint8_t dec(uint8_t b){ return b ^ ENC_KEY_CURRENT; }

#define ADC_CHANNEL_POT   3
```

```
24
25 // Map 0..1023 -> 0..7
26 static inline uint8_t channel_index_from_adc(uint16_t v){
27     uint16_t idx = v >> 7;              // divide by 128
28     if (idx >= NUM_CHANNELS) idx = NUM_CHANNELS-1;
29     return (uint8_t)idx;
30 }
31
32 // Read pot, pick channel, apply baud & key, show on LCD if changed
33 static uint8_t g_current_channel = 0xFF;   // invalid to force first
       update
34 void Channel_SelectAndApply(uint8_t show_on_lcd){
35     uint16_t raw = ADC_Read(ADC_CHANNEL_POT);
36     uint8_t idx  = channel_index_from_adc(raw);
37
38     if (idx != g_current_channel) {
39         g_current_channel = idx;
40         ENC_KEY_CURRENT   = CHANNELS[idx].key;
41         USART_Init(CHANNELS[idx].baud);
42         if (show_on_lcd){
43             LCD_cmd(0x01);
44             LCD_write('C'); LCD_write('H');
45             LCD_write('0' + idx);   // CH#
46         }
47     }
48 }
```

14 Send/Receive Path (SYNC, Encryption, LCD, LEDs)

14.1 Purpose & Requirements Mapping

This section implements the core link behavior:

- Transmitter computes baud from the selected channel, sends a fixed `SYNC` byte, then the message character.
- Receiver verifies the `SYNC` byte; on success shows green LED, decrypts the payload, maps it to a phrase, and prints on LCD; on failure shows red LED.
- Supports bonus "encrypt before sending, decrypt at receiver" via a simple XOR cipher.

14.2 How It Works

Baud selection. Each call to `sendMessage`/`receiveMessage` reads the potentiometer (`select_channel()`) and re-initializes the USART to that baud. (If you also use the channel table auto-apply function, keep these consistent so both ends stay tuned.)

Framing. Every packet is exactly two bytes:

1. **SYNC**: `0xAA` (sent in clear).
2. **Payload**: one ASCII code (`'H','O','R','S','A','B','Y'`) encrypted with `enc()`.

Receive path. The receiver reads the first byte and compares to `SYNC`.

- If equal: turns **green LED** on briefly, reads the encrypted payload, decrypts with `dec()`, selects the matching message string, clears the LCD and prints it.
- Else: blinks **red LED** to indicate a channel/baud mismatch or framing error.

Both branches end with a short delay and `LED_Off()`.

14.3 Verification

- With both nodes on the same channel, transmit each code and confirm the receiver shows the expected phrase and a green blink.
- Move one node to a different channel: the receiver should blink red on the first byte (wrong SYNC or framing).
- Enable/disable per-channel encryption and verify that mismatched keys produce unreadable payloads even when baud matches.

14.4 Code

Listing 8: Two-byte framed send/receive with SYNC check, LEDs, LCD, and XOR encryption

```
void sendMessage(char code) {
    unsigned long baud_rate = select_channel();
    USART_Init(baud_rate);

    USART_Transmit(SYNC_CODE);            // SYNC stays in clear
    USART_Transmit(enc((uint8_t)code));   // payload is encrypted
}

void receiveMessage(void) {
    unsigned long baud_rate = select_channel();
    USART_Init(baud_rate);

    uint8_t sync_byte = USART_Receive();

    if (sync_byte == SYNC_CODE) {
        LED_Green();

        uint8_t data_enc = USART_Receive();
        uint8_t data     = dec(data_enc);      // decrypt

        const char *msg = NULL;
        switch (data) {
            case 'H': msg = msg_H; break;
            case 'O': msg = msg_O; break;
            case 'R': msg = msg_R; break;
            case 'S': msg = msg_S; break;
            case 'A': msg = msg_A; break;
            case 'B': msg = msg_B; break;
            case 'Y': msg = msg_Y; break;
            default:  msg = "";    break;
        }
        LCD_cmd(0x01);
        for (uint8_t i = 0; msg[i] != '\0'; i++) LCD_write(msg[i]);

```

```
35          _delay_ms(200);
36          LED_Off();
37      } else {
38          LED_Red();
39          _delay_ms(200);
40          LED_Off();
41      }
42  }
```

15 Button Handling: Debounce and Long-Press Detection

15.1 Purpose & Requirements Mapping

The project requires three user buttons to send predefined messages. This helper adds reliable **debounce** and **long-press** detection so that each button can send two codes (short vs. long press), increasing the message set without extra hardware.

15.2 How It Works

`read_press_with_long(pin, pinreg)`:

1. Detects an *active-low* press and waits ∼18 ms for mechanical debounce.
2. Measures hold time in 10 ms ticks up to ∼1.2 s.
3. Classifies the press: `PRESS_SHORT` if < 600 ms, else `PRESS_LONG`.
4. Applies a release debounce before returning.

This allows mappings like:

$$\text{PD2: short} \rightarrow \text{'H'}, \quad \text{PD2: long} \rightarrow \text{'A'}$$

(and similarly for other buttons), doubling the available messages.

15.3 Verification

Hold a button for < 600 ms and confirm the short-press code is sent; hold for ≥ 600 ms to confirm the long-press code is sent. Scope the pin or add temporary LCD prints to visualize press type during testing.

15.4 Code

Listing 9: Blocking debounce + long-press (active-low buttons)

```
1  typedef enum { PRESS_NONE=0, PRESS_SHORT=1, PRESS_LONG=2 } press_t;
2
3  // Debounced read + long-press detect (blocking ~ up to 800ms max)
4  static press_t read_press_with_long(uint8_t pin, volatile uint8_t *
       pinreg_is_PIND){
5      // 1) wait for press (active low) with debounce
6      if ((*pinreg_is_PIND & (1<<pin)) == 0){
7          _delay_ms(18);
8          if ((*pinreg_is_PIND & (1<<pin)) != 0) return PRESS_NONE;
9
10         // 2) measure hold time in 10ms ticks
```

```
11          uint16_t ticks = 0;                  // 10ms units
12          while ((*pinreg_is_PIND & (1<<pin)) == 0 && ticks < 120){ //
                cap ~1.2s
13              _delay_ms(10);
14              ticks++;
15          }
16          // 3) release debounce
17          _delay_ms(18);
18          return (ticks >= 60) ? PRESS_LONG : PRESS_SHORT;  // 60*10ms =
                600ms
19      }
20      return PRESS_NONE;
21  }
```

16 How to Build & Run

1. Place `main.cpp` and this `.tex` alongside the PNGs.
2. Program both nodes with the same firmware.
3. In Proteus, connect UART Tx/Rx between nodes; ensure the pot feeds `ADC3`.
4. Power up: each node shows its channel. Use the Mode button to choose Tx/Rx.
5. Press buttons to transmit; verify LCD text and LED color on the peer.

17 Main Loop, Pins, and Mode Logic

17.1 Purpose & Requirements Mapping

This section ties all subsystems together and implements these requirements:

- **Tx/Rx mode selection**: user can switch the device between Sender and Receiver.
- **Three buttons send predefined messages**: short/long press doubles the set.
- **Channelization**: both ends must stay on the same channel (baud + key) to talk.
- **Stand-By heartbeat**: transmitter auto-sends `'S'` after inactivity.
- **Sleep after activity**: device idles (`IDLE`) until the next event.

17.2 How It Works

Pin configuration. LCD control on `PA0..PA2`, LCD data on `PORTC`, LEDs on `PB4`/`PB5`, UART `RX=PD0`/`TX=PD1`. Buttons are *inputs with pull-ups*: `PD2` (`'H'`/`'A'`), `PD3` (`'O'`/`'B'`), `PB2` (`'R'`/`'Y'`), and `PD5` for **mode toggle**.

Initialization. After enabling external interrupts and `sei()`, the code initializes LCD, ADC, selects and *applies* the current channel (sets baud & key and shows `CH#`), turns LEDs off, initializes USART (once), and starts Timer1 at 1 Hz for the heartbeat.

Mode toggle (PD5). A simple edge check with 50 ms debounce flips `mode` between Rx (0) and Tx (1) and updates the LCD (`CH# Tx/Rx`).

Stand-By heartbeat (Tx only). When `send_standby_flag` is set by Timer1 ISR, the loop (in Tx mode) blinks green, writes `'S'` locally, sends `'S'` (framed + encrypted), then resets counters. *Note:* with a 1 Hz timer, the current threshold `>=3` means ∼3 s; set `>=60` for ∼1 min.

Tx path. Reads the pot via `Channel_SelectAndApply(0)` to keep baud/key in sync. Each button uses `read_press_with_long()` to choose the code (short vs. long), sends it via `sendMessage()`, echoes the character on LCD, resets the heartbeat counter, and enters `IDLE` sleep.

Rx path. Also calls `Channel_SelectAndApply(0)` so baud/key track the pot. If data is waiting (`RXC`), it calls `receiveMessage()` (SYNC check, decrypt, LEDs, LCD) and enters `IDLE` sleep.

17.3 Verification

- Toggle to **Tx**, press each button (short/long) and confirm the peer displays the correct phrase and blinks green.
- Move either pot into a different channel band; the peer should blink red on SYNC and show no message.
- Leave the device idle in Tx; after the configured interval it should auto-send `'S'` and the peer should display `Stand By XD`.
- In both modes, observe current draw or use Proteus to verify the MCU sleeps between events.

17.4 Code

Listing 10: Main loop: pins, mode toggle, heartbeat, Tx/Rx paths

```
#define BTN_MODE_PIN    PD5
#define BTN_H_PIN       PD2    // send 'H'
#define BTN_O_PIN       PD3    // send 'O'
#define BTN_R_PIN       PB2    // send 'R'

static inline uint8_t is_low_PIND(uint8_t pin) { return !(PIND & (1 << pin)); }
static inline uint8_t is_low_PINB(uint8_t pin) { return !(PINB & (1 << pin)); }

static inline void lcd_show_mode(uint8_t mode_tx) {
    LCD_cmd(0x01); // clear
    if (mode_tx) { LCD_write('T'); LCD_write('x'); }
    else         { LCD_write('R'); LCD_write('x'); }
}

static inline void lcd_show_mode_and_ch(uint8_t mode_tx){
    LCD_cmd(0x01);
    LCD_write('C'); LCD_write('H'); LCD_write('0' + g_current_channel);
    LCD_write(' '); LCD_write(mode_tx ? 'T' : 'R'); LCD_write('x');
}

int main(void)
{
    // LCD control pins on PORTA
    DDRA |= (1 << PA0) | (1 << PA1) | (1 << PA2);
    DDRC  = 0xFF;                  // LCD data on PORTC
    DDRB |= (1 << PB4) | (1 << PB5); // LEDs
```

```
27
28      // USART pins
29      DDRD &= ~(1 << PD0);           // RX input
30      DDRD |=  (1 << PD1);           // TX output
31
32      // Buttons as inputs + pull-ups
33      DDRD &= ~((1 << BTN_H_PIN) | (1 << BTN_O_PIN) | (1 << BTN_MODE_PIN)
            );
34      DDRB &= ~(1 << BTN_R_PIN);
35      PORTD |= (1 << BTN_H_PIN) | (1 << BTN_O_PIN) | (1 << BTN_MODE_PIN);
             // pull-ups
36      PORTB |= (1 << BTN_R_PIN);
                                                        // pull-up
37
38      // External interrupts (INT0, INT1 on falling edge). INT2 is
            enabled as before.
39      MCUCR |= (1 << ISC01) | (1 << ISC11);
40      GICR  |= (1 << INT0) | (1 << INT1) | (1 << INT2);
41
42      sei();                // global interrupts
43
44      init_LCD();
45      ADC_Init();
46      Channel_SelectAndApply(1);  // sets baud+key and shows "CH#"
47      LED_Off();
48
49      uint8_t mode = 0;     // 0 = Receiver (default), 1 = Sender
50      uint8_t lastButtonState = (PIND & (1 << BTN_MODE_PIN));
51
52      unsigned long baud = select_channel();
53      USART_Init(baud);
54
55      Timer1_Init();        // Initialize timer (currently ~3 s threshold
            in ISR)
56
57      while (1)
58      {
59          // --- Mode toggle button check (PD5) ---
60          uint8_t currentButtonState = (PIND & (1 << BTN_MODE_PIN));
61          if (lastButtonState != currentButtonState) {
62              _delay_ms(50); // debounce
63              if (is_low_PIND(BTN_MODE_PIN)) { // pressed
64                  mode ^= 1;                   // toggle
65                  lcd_show_mode_and_ch(mode);
66              }
67          }
68          lastButtonState = currentButtonState;
69
70          // --- Standby flag handling (Tx only) ---
71          if (send_standby_flag && mode) {
72              send_standby_flag = 0;
73
74              LED_Green();
75              LCD_write('S');
76
77              sendMessage('S');   // already encrypts & uses current
                    channel
78
```

```
79              _delay_ms(90);
80              LED_Off();
81              standby_counter = 0;
82          }
83
84          if (mode == 1) { // Sender
85              Channel_SelectAndApply(0);
86
87              press_t p;
88
89              // PD2 -> 'H' (short) / 'A' (long)
90              p = read_press_with_long(BTN_H_PIN, &PIND);
91              if (p){
92                  char c = (p == PRESS_LONG) ? 'A' : 'H';
93                  sendMessage(c); LCD_write(c); standby_counter = 0;
                        go_to_sleep();
94              }
95
96              // PD3 -> 'O' (short) / 'B' (long)
97              p = read_press_with_long(BTN_O_PIN, &PIND);
98              if (p){
99                  char c = (p == PRESS_LONG) ? 'B' : 'O';
100                 sendMessage(c); LCD_write(c); standby_counter = 0;
                        go_to_sleep();
101             }
102
103             // PB2 -> 'R' (short) / 'Y' (long)
104             p = read_press_with_long(BTN_R_PIN, &PINB);
105             if (p){
106                 char c = (p == PRESS_LONG) ? 'Y' : 'R';
107                 sendMessage(c); LCD_write(c); standby_counter = 0;
                        go_to_sleep();
108             }
109         }
110         else {          // Receiver
111             Channel_SelectAndApply(0);    // keeps baud+key in sync
112
113             if (UCSRA & (1 << RXC)) {
114                 receiveMessage();
115                 go_to_sleep();
116             }
117         }
118     }
119 }
```

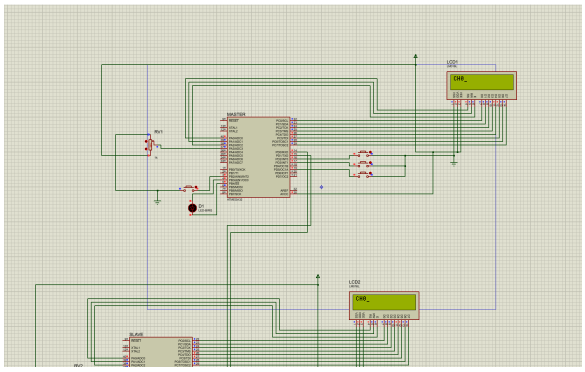
Notes. (1) To match a **1-minute** heartbeat, change the ISR threshold to `>=60`. (2) If you use both `select_channel()` and `Channel_SelectAndApply()`, ensure they select the same baud/key mapping to avoid drift.

18 Startup & Mode Toggle (Step 0)

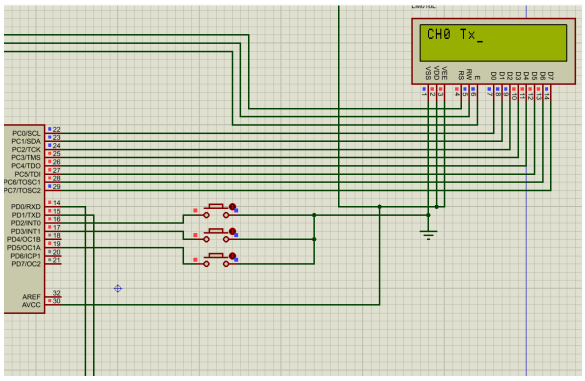
18.1 What You See at Startup

On power-up, each node initializes the LCD and immediately shows the current channel as `CH#`. The default mode is **Rx**. The user toggles the mode with the **PD5 / OC1A** button: **press**

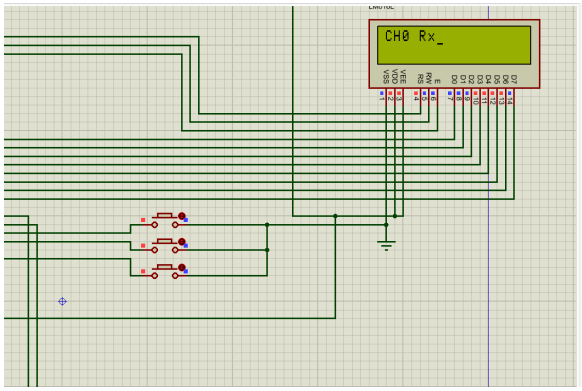
and hold to switch to **Tx**, press and hold again to switch back to **Rx**. The LCD always shows `CH# Tx` or `CH# Rx` after a successful toggle.



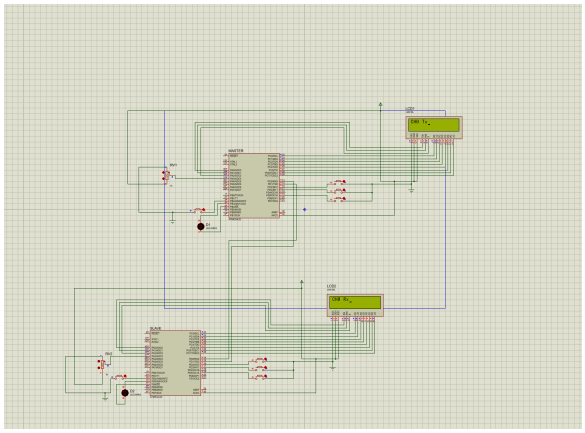
(a) Both nodes powered, channel shown.



(b) Tx node after long-press on PD5: `CH0 Tx`.



(c) Rx node after next long-press: `CH0 Rx`.



(d) Overview — Master in Tx, Slave in Rx.

Figure 4: Startup and mode toggle (PD5 long-press).

Requirement(s) covered (English). "User can select Tx/Rx mode; LCD must reflect the mode and channel."

18.2 Code change: require *long-press* on PD5 to toggle mode

Replace the previous edge-based toggle with this long-press check:

Listing 11: Mode toggle: PD5 long-press

```
/* inside while(1) loop, replace the old PD5 toggle block with: */
press_t mp = read_press_with_long(BTN_MODE_PIN, &PIND);
if (mp == PRESS_LONG) {                    // require hold to toggle
    mode ^= 1;                             // 0=Rx, 1=Tx
    standby_counter = 0;                   // reset heartbeat timer
    lcd_show_mode_and_ch(mode);
    go_to_sleep();                         // small idle after toggle
    continue;                              // skip to next loop iteration
}
```

19 Scenario 1 — Correct Stand-By (green LED, 1-minute idle)

19.1 Objective

Show the automatic **Stand-By** transmission after **1 minute** of no user activity in **Tx** mode, with successful reception indicated by the **green LED** and the LCD phrase "Stand By XD".

19.2 Setup

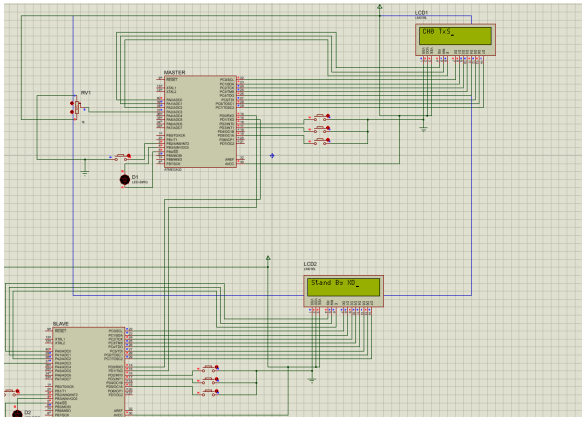
Both nodes on the same channel (e.g., CH0). Transmitter in **Tx** mode, receiver in **Rx** mode. Per-channel encryption enabled on both ends.

19.3 Steps

1. On the transmitter, perform no button presses for ≈ 60 s.
2. When the heartbeat timer expires, the Tx sends a two-byte frame: `SYNC=0xAA` then encrypted payload `'S'`.
3. The receiver checks `SYNC`, blinks **green**, decrypts, maps `'S'` to `"Stand By XD"`, and prints it on the LCD.

19.4 Expected Output

Receiver's LCD shows `Stand By XD`; green LED briefly on; both nodes return to `IDLE` sleep afterwards.



(a) Tx waiting idle on `CH0 Tx.` - Rx shows `Stand By XD` after SYNC match (green LED).

Figure 5: Stand-By heartbeat success after 1 minute of inactivity.

19.5 Code change: make the heartbeat truly ∼1 minute

Update the Timer1 compare ISR threshold from 3 to 60 ticks (1 Hz):

Listing 12: 1-minute heartbeat

```
ISR(TIMER1_COMPA_vect) {
    standby_counter++;
    if (standby_counter >= 60) {    // ~60 s at 1 Hz
        send_standby_flag = 1;
    }
}
```

Where the requirement comes from (English). "Send a periodic 'Stand By' message (about 1 minute) to verify both devices are within range and on the same channel; receiver should indicate success with the green LED."

20 Scenario 2 — Six User Messages (H, O, R, A, B, Y)

20.1 Objective

Verify that all six user messages (three buttons, short/long presses) are transmitted with SYNC, decrypted correctly at the receiver, indicated by the **green LED**, and shown on the LCD as the intended phrases.

20.2 Setup

Both nodes on the same channel (e.g., CH0). Transmitter in **Tx** mode, receiver in **Rx** mode. Per-channel encryption keys match.

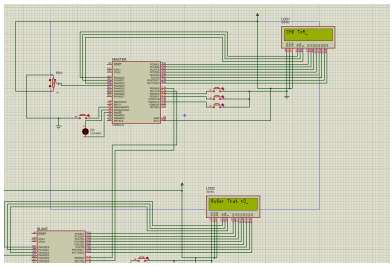
20.3 Steps

1. **PD2**: short press → `'H'` ("HELP :()"), long press → `'A'` ("ACK :)").
2. **PD3**: short press → `'O'` ("OVER :)"), long press → `'B'` ("BUSY :|").
3. **PB2**: short press → `'R'` ("Roger That :D"), long press → `'Y'` ("YES!").

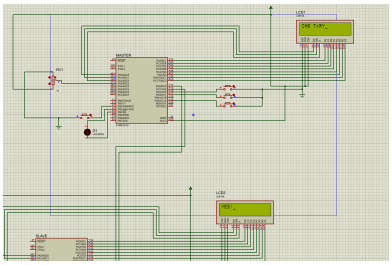
20.4 Expected Output

For each press the receiver observes `SYNC=0xAA` followed by the (XOR)-encrypted payload, lights the **green LED** on SYNC match, decrypts the payload, and prints the mapped phrase on its LCD. The transmitter echoes the sent character on its LCD.

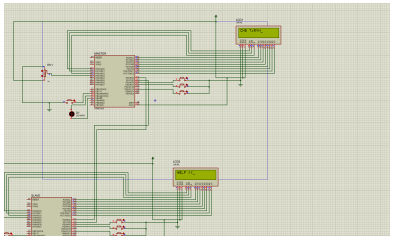
20.5 Evidence (Proteus captures)



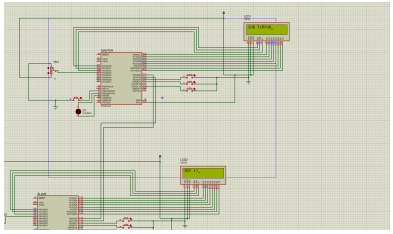
(a) **R** — *Roger That :D*



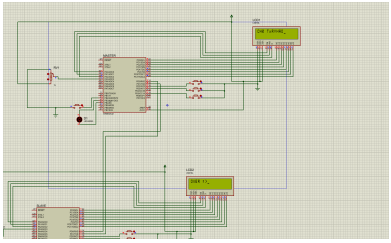
(b) **Y** — *YES!*



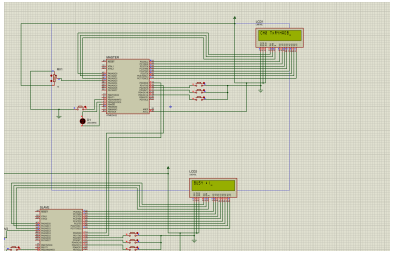
(c) **H** — *HELP :(*



(d) **A** — *ACK :)*



(e) **O** — *OVER :)*



(f) **B** — *BUSY :|*

Figure 6: Receiver LCD confirms correct decode for all six user messages.

20.6 Requirement(s) covered (English)

- "Each device has three buttons; each button sends a character. The receiver shows the corresponding phrase on the LCD."
- "A fixed sync character is sent before the message so the receiver can verify communication (green on success)."
- "(Bonus) Encrypt before sending and decrypt at the receiver."

20.7 (Optional) Test helper to reproduce this scenario quickly

Listing 13: Demo sender: send all six user messages

```
static void demo_send_all_six(void){
    const char seq[] = { 'H','A','O','B','R','Y' }; // short/long pairs
        grouped
    for (uint8_t i = 0; i < sizeof(seq); i++){
        Channel_SelectAndApply(0);     // keep baud/key synced to pot
        sendMessage(seq[i]);
        LCD_write(seq[i]);             // local echo
        standby_counter = 0;           // reset heartbeat timer
        _delay_ms(500);
    }
}
```

21 Scenario 3 — Channel Mismatch (red LED, no LCD update)

21.1 Objective

Demonstrate that when the transmitter and receiver are on **different channels** (different baud/key), the receiver fails the SYNC check, blinks the **red LED**, and does not print any message on the LCD.

21.2 Setup

Transmitter set to **CH0** (e.g., 115200 baud), receiver set to **CH5** (e.g., 76800 baud). Mode: Tx on the sender, Rx on the receiver.

21.3 Steps

1. On the transmitter (CH0), press the PD2 button to send code `'H'`.
2. The receiver (CH5) attempts to read the first byte at its own baud/key.

21.4 Expected Output

Because the channels (baud/key) don't match, the first byte is not recognized as `SYNC=0xAA`. The receiver immediately indicates failure by lighting the **red LED** briefly and **no text** is written to the LCD.

21.5 Requirement(s) covered (English)

- "Use a potentiometer to select the channel; each channel has a different baud rate; if devices are on different channels they cannot read each other's messages."
- "LEDs indicate link status: green on success, red on failure."

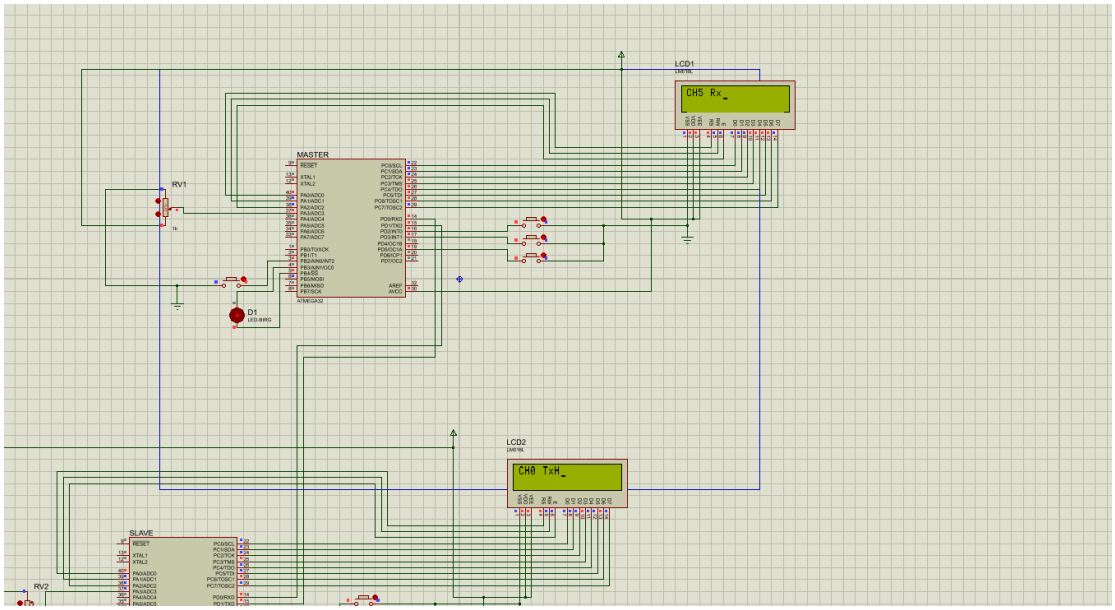


Figure 7: Mismatch example: Rx on `CH5` and Tx on `CH0`. Pressing `'H'` on Tx causes a red LED on Rx and no LCD message.

21.6 Code path for this behavior

No change is required—the existing receive routine already handles it. The `else` branch below is the one exercised when `SYNC` is not detected (because of baud/key mismatch):

Listing 14: SYNC-fail branch on channel mismatch (red LED, no LCD text)

```
void receiveMessage(void) {
    unsigned long baud_rate = select_channel();
    USART_Init(baud_rate);

    uint8_t sync_byte = USART_Receive();

    if (sync_byte == SYNC_CODE) {
        // ... success path (green, decrypt, print)
    } else {
        LED_Red();            // indicate failure
        _delay_ms(200);
        LED_Off();            // no LCD update on error
    }
}
```

22 Scenario 4 — Power ON/OFF via PD4 (OC1B)

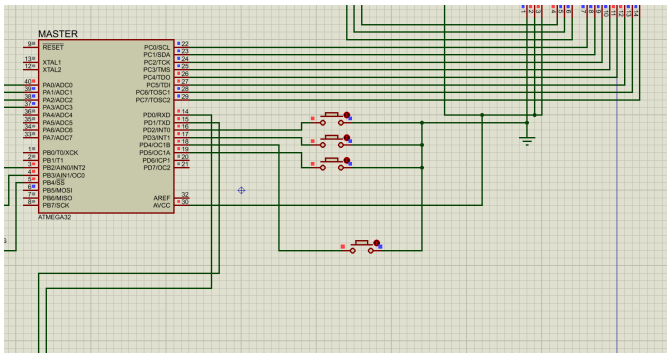
22.1 Objective

Add a dedicated power control: pressing the **PD4/OC1B** button toggles the node between **ON** and **OFF**. In OFF state we shut down Timer1, ADC, and USART, turn LEDs off, and show `OFF` on the LCD; pressing again returns to normal operation (channel & key applied, heartbeat restarted).

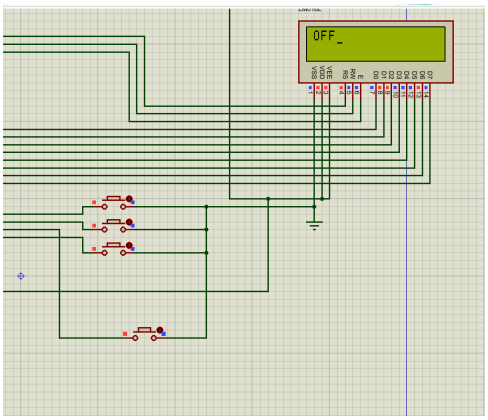
22.2 How It Works

- **Helpers**: `Periph_Stop()` disables Timer1 (prescaler+IRQ), ADC, and USART, turns LEDs off, and clears the LCD. `Periph_Start(mode)` re-initializes ADC, reapplies channel (baud+key), restarts Timer1, and shows `CH# Tx/Rx`.
- **Button PD4**: input with pull-up; we debounce and detect the *falling edge* (press). Because ATmega32 has no pin-change IRQs on PD4, OFF mode uses a tiny polling delay (∼20 ms) to catch the next press.

22.3 Evidence



(a) Power key wiring on PD4/OC1B (pull-up).



(b) OFF state on the LCD; comms/heartbeat disabled.

Figure 8: Power ON/OFF scenario.

22.4 Code

Listing 15: Power control: helpers and globals

1) Power helpers (top of file with other globals).

```
#define BTN_PWR_PIN    PD4                    // On/Off button on OC1B pin
volatile uint8_t system_on = 1;              // start ON

// Stop peripherals to save power while "OFF"
static inline void Periph_Stop(void){
    LED_Off();
    // Stop Timer1 and its interrupt
    TCCR1B &= ~((1<<CS12)|(1<<CS11)|(1<<CS10));
    TIMSK  &= ~(1<<OCIE1A);
    // Disable ADC
    ADCSRA &= ~(1<<ADEN);
    // Disable USART (and RX IRQ)
    UCSRB &= ~((1<<RXEN)|(1<<TXEN)|(1<<RXCIE));
    // Quiet the LCD (optional clear)
    LCD_cmd(0x01);
}

// Restart peripherals after turning ON
static inline void Periph_Start(uint8_t mode_tx){
    ADC_Init();
    Channel_SelectAndApply(1);    // sets baud+key & shows CH#
```

```
    Timer1_Init();                // re-enables TIMSK OCIE1A
    lcd_show_mode_and_ch(mode_tx);
}
```

Listing 16: PD4 as input with pull-up

2) Init pins (add to `main()` init).

```
DDRD &= ~(1 << BTN_PWR_PIN);      // PD4 input
PORTD |= (1 << BTN_PWR_PIN);      // pull-up enabled
```

Listing 17: Edge-detected toggle + OFF polling

3) Main loop integration (put this at the very top of the `while(1)` body).

```
/* --- Power button (PD4) edge detection --- */
static uint8_t lastPowerPin = 1;                    // pull-up idle = 1
uint8_t nowPowerPin = (PIND >> BTN_PWR_PIN) & 1;

if (lastPowerPin != nowPowerPin) {
    _delay_ms(20);                                  // debounce
    nowPowerPin = (PIND >> BTN_PWR_PIN) & 1;
    if (lastPowerPin == 1 && nowPowerPin == 0) { // falling edge =
        press
        system_on ^= 1;
        if (!system_on) {
            // Going OFF
            Periph_Stop();
            LCD_cmd(0x01); LCD_write('O'); LCD_write('F'); LCD_write('F
                ');
        } else {
            // Going ON
            Periph_Start(mode);  // 'mode' is your Tx/Rx flag
        }
    }
}
lastPowerPin = nowPowerPin;

// In OFF state, keep a light polling loop so PD4 can be read again
if (!system_on) {
    _delay_ms(20);          // light idle; keep CPU alive to poll PD4
    continue;               // skip the rest of the loop while OFF
}
```

23 Conclusion

The project demonstrates a compact digital "walkie-talkie" over UART with channelization, simple per-channel encryption, and a usable UI. The design is portable to real hardware with minor baud choices and crystal considerations.