You're absolutely right — let's complete the detailed explanation of **Project 7: Visitor Counter using IR Sensors and LCD** from where we left off.

🔧 Project 7: Visitor Counter using IR Sensors and LCD

🎯 Objective:

To count how many people **enter** or **exit** a room using **two IR sensors**, and display the count live on an **LCD**.

🛠️ Hardware:

- STM32F103C6/C8
- 2 IR sensors:
  - **Entry sensor** on PA0 (active LOW)
  - **Exit sensor** on PA1 (active LOW)
- 16x2 LCD connected to:
  - LCD_RS → PB1
  - LCD_EN → PB2
  - LCD_D4–D7 → PB12 to PB15

🧠 Code Breakdown (Continued & Detailed):

📌 LCD Pin Assignment

```
sbit LCD_RS at GPIOB_ODR.B1;

sbit LCD_EN at GPIOB_ODR.B2;

sbit LCD_D4 at GPIOB_ODR.B12;

sbit LCD_D5 at GPIOB_ODR.B13;

sbit LCD_D6 at GPIOB_ODR.B14;
```

sbit LCD_D7 at GPIOB_ODR.B15;

This maps each LCD control/data line to a specific pin on GPIOB.

📌 Global Variables

unsigned int peopleCount = 0;

char txt[7];

- peopleCount: keeps track of number of people inside.
- txt: buffer to hold string for displaying the number.

🧩 NumberToStr() Function

```
void NumberToStr(unsigned int num, char *output) {
  char temp[7];
  int i = 0, j = 0;

  if (num == 0) {
    output[0] = '0';
    output[1] = '\0';
    return;
  }

  while (num > 0) {
    temp[i++] = (num % 10) + '0';  // Get digits in reverse order
    num /= 10;
  }

  while (i > 0) {
```

```
    output[j++] = temp[--i];      // Reverse the string
  }
  output[j] = '\0';
}
```

Purpose: Converts a number like 25 into a string "25" manually (without using built-in IntToStr()).

🧩 UpdateLCD() Function

```
void UpdateLCD() {
  Lcd_Out(2, 8, "      ");  // Clear previous digits on LCD
  NumberToStr(peopleCount, txt);
  Lcd_Out(2, 8, txt);       // Display updated count
}
```

Purpose: Updates only the number area of the LCD to avoid redrawing the whole screen. Keeps the interface clean.

🧩 main() Function

```
void main() {
  // Initialize LCD
  Lcd_Init();
  Lcd_Cmd(_LCD_CLEAR);
  Lcd_Cmd(_LCD_CURSOR_OFF);
  Lcd_Out(1, 1, "People Counter");
  Lcd_Out(2, 1, "Count: 0");
```

Initializes the LCD and prints the static headers.

```
  // Set PA0 and PA1 as inputs with pull-up
```

```
GPIOA_CRL = 0x00000088;      // PA0, PA1 as input with pull-up/down
GPIOA_BSRR = 0x00000003;     // Pull-up both PA0 and PA1
```

- **0x88** sets PA0 and PA1 to input.
- BSRR sets the pull-up (logic HIGH default). The IR sensor pulls them LOW when activated.

🔁 Main Loop: Detect Entry & Exit

```
while(1) {
  if (!GPIOA_IDR.B0) {       // Entry detected (PA0 goes LOW)
    peopleCount++;           // Increase count
    UpdateLCD();             // Update LCD
    Delay_ms(500);           // Debounce delay
  }

  if (!GPIOA_IDR.B1) {       // Exit detected (PA1 goes LOW)
    if (peopleCount > 0)     // Prevent negative count
      peopleCount--;
    UpdateLCD();
    Delay_ms(500);
  }

  Delay_ms(10);              // Loop debounce/stability
 }
}
```

📌 Logic Summary:

Sensor Triggered	Action	Condition
PA0 (Entry IR)	peopleCount++	Always
PA1 (Exit IR)	peopleCount--	Only if > 0

- A 500ms delay prevents bouncing or false triggers.
- Count is shown in real time on the LCD.

🧠 **What You Learn from This Project:**

- How to use **digital inputs** to monitor physical events.
- How to **interface an LCD** with an STM32.
- How to **convert numbers to strings manually**.
- How to implement a **basic queue counter logic** (entry/exit system).
- Use of **debounce delay** for IR-based inputs.

Would you like a Proteus circuit schematic for this project or suggestions for enhancements like buzzer alerts or EEPROM storage for visitor history?

Ready to continue with detailed **Project 8** again or move to another?

Perfect! Let's now proceed with a **detailed explanation of Project 8** as promised.

🔧 **Project 8: Soil Moisture Monitoring & Pump Control**

🎯 **Objective:**

To monitor soil moisture using an analog sensor and control:

- a **pump** (via a relay)
- an **LED indicator**

...based on whether the moisture level falls **below 50%**.

🛠️ **Hardware:**

- **Soil moisture sensor** (analog output connected to PA0)
- **LED** connected to PB1
- **Relay (pump control)** connected to PB2
- STM32F103C6/C8 microcontroller

📄 **MikroC Code (Reviewed & Explained):**

#define LED GPIOB_ODR.B1

#define relay GPIOB_ODR.B2

🔹 **Defines:**

Aliases for GPIO pins to make code readable:

- LED is on PB1
- Relay is on PB2

🔁 Main Program

```
void main() {
  unsigned int adc_value;
  float moisture_percent;

  // Configure PB0, PB1, PB2 as digital outputs
  GPIO_Config(&GPIOB_BASE,
    _GPIO_PINMASK_0 | _GPIO_PINMASK_1 | _GPIO_PINMASK_2,
    _GPIO_CFG_MODE_OUTPUT | _GPIO_CFG_SPEED_MAX | _GPIO_CFG_OTYPE_PP);

  // Configure PA0 as analog input (moisture sensor input)
  GPIO_Config(&GPIOA_BASE, _GPIO_PINMASK_0, _GPIO_CFG_MODE_ANALOG);

  // Initialize ADC
  ADC1_Init();

  // Initially turn off LED and relay
  LED = 0;
  relay = 0;

  while(1) {
    // Read analog sensor value
    adc_value = read_ADC();

    // Convert ADC (0–4095) to percentage
    moisture_percent = ((float)adc_value / 4095.0f) * 100.0f;
```

```
    if (moisture_percent < 50.0f) {
      // Soil is dry → Turn ON pump and LED
      LED = 1;
      relay = 1;
    } else {
      // Soil is moist → Turn OFF pump and LED
      LED = 0;
      relay = 0;
    }

    Delay_ms(100);  // Short delay between readings
  }
}
```

🧪 Supporting Functions (ADC)

🔧 read_ADC()

```
unsigned int read_ADC() {
  ADC1_CR2 |= (1 << 22);         // Start conversion (SWSTART bit)
  while (!(ADC1_SR & (1 << 1)));   // Wait for EOC (End Of Conversion)
  return ADC1_DR;                // Return 12-bit ADC result
}
```

🔧 ADC1_Init()

```
void ADC1_Init() {
  RCC_APB2ENR |= (1 << 9);       // Enable ADC1 clock
  ADC1_CR2 = 0;                  // Reset control register 2
```

```
  ADC1_CR1 = 0;              // Reset control register 1
  ADC1_SMPR2 |= (7 << 0);      // Max sample time on channel 0 (PA0)
  ADC1_CR2 |= (1 << 0);        // Turn ON ADC
  Delay_ms(1);               // Stabilize
}
```

📊 **How ADC Converts Moisture to Percent:**

- **ADC range**: 0 to 4095 (12-bit)
- If moisture sensor gives **low voltage**, soil is dry → **low ADC value**
- If it gives **high voltage**, soil is wet → **high ADC value**

So:

moisture_percent = (adc_value / 4095.0) * 100

is used to scale the sensor reading to a **0%–100% scale**.

💡 **Control Logic:**

Moisture %	LED	Pump (Relay)
< 50%	ON	ON
>= 50%	OFF	OFF

📌 **Engineering Concepts Learned:**

Topic	Application
GPIO Output	Controls LED and Relay
GPIO Analog Input	Reads soil sensor via ADC (PA0)
ADC (12-bit)	Converts analog voltage to digital moisture value

Topic	Application
Voltage to Percentage	Sensor value is scaled to human-readable percentage
Relay Activation	Turns on a pump or motor in real-world use

⚙️ **Potential Enhancements:**

- Add **LCD** to show exact moisture %.
- Use EEPROM to **log moisture readings**.
- Add **WiFi or Bluetooth module** to **transmit soil data** to a phone or dashboard.
- Add **a manual override button** to trigger pump regardless of sensor reading.

Let me know if you'd like:

- A **Proteus simulation diagram** for this project
- Or should we now move to **Project 9: Ultrasonic Distance Meter + LCD** with the same level of detail?

Excellent! Let's proceed with a **deep dive into Project 9: Ultrasonic Distance Meter using HC-SR04 & LCD**.

🔧 Project 9: Ultrasonic Distance Meter with LCD

🎯 Objective:

Measure the distance to an object using an **HC-SR04 ultrasonic sensor** and display the result in centimeters on a **16x2 LCD**.

🛠️ Hardware Used:

- STM32F103C6 or C8 (ARM Cortex-M3)
- **Ultrasonic Sensor** (HC-SR04):
  - **TRIG**: PA1 (output)
  - **ECHO**: PA0 (input)
- **16x2 LCD**:
  - Connected in **4-bit mode** using GPIOB (PB0–PB5)
- **Timer2**: Used to measure echo pulse width in microseconds

---

📄 Understanding HC-SR04:

Pin	Function
TRIG	Sends a 10µs pulse to start measurement
ECHO	Goes HIGH for a duration proportional to distance (in µs)

📐 Distance Formula:

$$\text{Distance (cm)} = \text{Duration (µs)} \times 0.03432 \approx \text{Duration} \times 0.017$$

$$\text{Distance (cm)} = \frac{\text{Duration (µs)} \times 0.0343}{2} \approx \text{Duration} \times 0.017$$

📜 Full Code Breakdown (MikroC)

1️⃣ LCD Control Definitions

```
#define LCD_RS GPIOB_ODR.B0
#define LCD_EN GPIOB_ODR.B1
#define LCD_D4 GPIOB_ODR.B2
#define LCD_D5 GPIOB_ODR.B3
#define LCD_D6 GPIOB_ODR.B4
#define LCD_D7 GPIOB_ODR.B5
```

Assigns the **LCD pins** to GPIOB outputs for custom control.

2️⃣ LCD Send Nibble / Command / Character

```
void Lcd_SendNibble(unsigned char nibble) {
  LCD_D4 = (nibble >> 0) & 1;
  LCD_D5 = (nibble >> 1) & 1;
  LCD_D6 = (nibble >> 2) & 1;
  LCD_D7 = (nibble >> 3) & 1;
  LCD_EN = 1;
  Delay_us(1);
  LCD_EN = 0;
  Delay_us(1);
}
```

- Sends a **4-bit value** to the LCD.
- Called twice per byte: upper nibble, then lower nibble.

3️⃣ LCD Command and Character Write

```
void Lcd_SendCommand(char cmd) {
  LCD_RS = 0;  // Command mode
  Lcd_SendNibble(cmd >> 4);
  Lcd_SendNibble(cmd);
  Delay_ms(2);
}
void Lcd_WriteChar(char row, char col, char chr) {
  char pos = (row == 2) ? 0xC0 : 0x80;
  Lcd_SendCommand(pos + col - 1);
  LCD_RS = 1;  // Data mode
  Lcd_SendNibble(chr >> 4);
  Lcd_SendNibble(chr);
}
```

4️⃣ LCD Initialization

```
void Lcd_Init_Custom() {
  GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_0 | _GPIO_PINMASK_1 |
                      _GPIO_PINMASK_2 | _GPIO_PINMASK_3 |
                      _GPIO_PINMASK_4 | _GPIO_PINMASK_5);

  LCD_RS = 0; LCD_EN = 0;
  Delay_ms(20);
  Lcd_SendNibble(0x03); Delay_ms(5);
  Lcd_SendNibble(0x03); Delay_us(150);
  Lcd_SendNibble(0x03); Delay_us(150);
```

```
  Lcd_SendNibble(0x02);  // Set 4-bit mode

  Lcd_SendCommand(0x28);  // Function Set: 2 line, 5x7 font
  Lcd_SendCommand(0x0C);  // Display ON, cursor OFF
  Lcd_SendCommand(0x06);  // Entry Mode
  Lcd_SendCommand(0x01);  // Clear
}
```

5️⃣ Ultrasonic Sensor TRIG & ECHO Handling

```
#define ECHO_IN  GPIOA_IDR.B0  // Input from sensor (ECHO)
#define TRIG_OUT GPIOA_ODR.B1  // Output to sensor (TRIG)
```

🔹 **trigger_and_measure()**

```
void trigger_and_measure() {
  GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_1);  // TRIG = output
  GPIO_Digital_Input(&GPIOA_BASE, _GPIO_PINMASK_0);   // ECHO = input

  TRIG_OUT = 0; delay_us(2);       // Ensure TRIG is low
  TRIG_OUT = 1; delay_us(10);      // Send 10µs pulse
  TRIG_OUT = 0;

  while (!ECHO_IN);              // Wait for ECHO to go HIGH
  TIM2_CNT = 0;                  // Reset timer

  while (ECHO_IN && TIM2_CNT < 60000); // Wait while ECHO is HIGH
  duration = TIM2_CNT;           // Read duration
}
```

6️⃣ Timer2 Setup

```
// Timer2: 1 tick = 1 µs
RCC_APB1ENR.TIM2EN = 1;    // Enable Timer2
TIM2_PSC = 72 - 1;         // Prescaler: 72MHz / 72 = 1MHz → 1 tick = 1µs
TIM2_ARR = 0xFFFF;         // Auto-reload max
TIM2_CR1.CEN = 1;          // Start timer
```

7️⃣ Main Logic

```
void main() {
  Lcd_Init_Custom();
  Lcd_WriteText(1, 1, "Ultrasonic Begin");
  Delay_ms(1000);
  Lcd_SendCommand(0x01); // Clear LCD

  // Timer2 config done here...

  while (1) {
    trigger_and_measure();          // Send pulse & measure echo
    distance = duration * 0.017;    // Convert to cm
    FloatToStr(distance, txt);      // Convert to string
    txt[5] = 0;                     // Truncate after 2 decimals

    Lcd_SendCommand(0x01);          // Clear LCD
    Lcd_WriteText(1, 1, "Distance:");
    Lcd_WriteText(2, 1, txt);
```

```
    Lcd_WriteText(2, 7, "cm");


    Delay_ms(500);
  }
}
```

🧠 **Key Engineering Concepts:**

Concept	Description
PWM pulse generation	TRIG pin sends a 10µs HIGH pulse
Echo pulse timing	Timer2 measures the HIGH pulse width of ECHO
Timer prescaling	Used to scale clock to microsecond resolution
Floating-point math	Converts time duration to real-world distance
4-bit LCD interfacing	Saves pins while maintaining display functionality
Polling technique	Waits for ECHO signal changes with while() loops

---

📏 **Example Calculation:**

If duration = 400 µs:

distance = 400 * 0.017 = 6.8 cm

🧩 **Limitations & Considerations:**

- Blocking while() loops (could miss rapid changes).
- Timer2 timeout is capped (~60ms = ~10 meters).
- **Float operations are heavy** on Cortex-M3 (could use integer math instead).

✅ **Possible Enhancements:**

- Add **threshold-based LED or buzzer alert** if object is too close.
- Use **interrupts** instead of polling for ECHO.
- Display **status messages** like “Too Far” or “Too Close”.

Let me know if you'd like:

- Proteus simulation circuit suggestions
- Conversion to STM32CubeIDE version (HAL drivers)
- Help with debugging a similar project

Would you like to proceed now with **Project 10: Weather Station using ADC and I2C**?

Great! Let's now dive into **Project 10: Weather Station using DHT11, LDR, and LCD** — one of the most integrated and sensor-rich projects in your embedded systems set.

🔧 Project 10: Weather Station using ADC and DHT11

🎯 Objective:

Build a simple weather station that:

- Measures **temperature and humidity** using the **DHT11** sensor (digital)
- Measures **light intensity** using an **LDR sensor** (analog via ADC)
- Displays all values on a **16x2 LCD screen**

🛠️ Hardware Used:

- STM32F103C6/C8 microcontroller
- **DHT11** sensor (digital one-wire protocol, connected to PA0)
- **LDR sensor** (connected to PA4 via voltage divider → ADC input)
- **16x2 LCD** (in 4-bit mode using PA1, PA2, PB0, PB1, PB10, PB11)

📜 Code Breakdown

1️⃣ LCD Pin Mapping

```
sbit LCD_RS at GPIOA_ODR.B1;

sbit LCD_EN at GPIOA_ODR.B2;

sbit LCD_D4 at GPIOB_ODR.B0;

sbit LCD_D5 at GPIOB_ODR.B1;

sbit LCD_D6 at GPIOB_ODR.B10;
```

sbit LCD_D7 at GPIOB_ODR.B11;

Maps each LCD signal to GPIO ports and pins for easy reference.

2️⃣ DHT11 Sensor Pin and Macros

#define DHT11_PIN GPIOA_IDR.B0

#define DHT11_OUT GPIOA_ODR.B0

#define DHT11_SET_OUT GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_0)

#define DHT11_SET_IN GPIO_Digital_Input(&GPIOA_BASE, _GPIO_PINMASK_0)

These macros help us:

- Switch **between input/output** dynamically for one-wire communication
- Read and write to **PA0**, the DHT11 data pin

3️⃣ Global Variables

unsigned char dht[5]; // Array to store 5 bytes from DHT11

char text[16]; // Buffer for displaying data

unsigned int light; // LDR ADC value

4️⃣ Microsecond Delay Function

void delay_us(unsigned int us) {

  while(us--) Delay_us(1);

}

Provides controlled short delays (used in DHT11 timing protocol).

5️⃣ DHT11 Communication

char DHT11_ReadByte()

char i, b = 0;

```
for(i = 0; i < 8; i++) {
   while(!DHT11_PIN);       // Wait for LOW-to-HIGH
   delay_us(30);            // Wait ~30us
   if(DHT11_PIN) b |= (1 << (7 - i));  // If still HIGH, it's a 1
   while(DHT11_PIN);        // Wait till signal goes LOW again
}
return b;
```

char DHT11_Read()

```
DHT11_SET_OUT;
DHT11_OUT = 0; Delay_ms(18);  // Pull LOW for at least 18ms
DHT11_OUT = 1; Delay_us(30);  // Pull HIGH briefly

DHT11_SET_IN;

if (DHT11_PIN) return 0;     // No response
while(!DHT11_PIN);          // Wait for LOW
while(DHT11_PIN);           // Wait for HIGH

for(i = 0; i < 5; i++) dht[i] = DHT11_ReadByte();  // Read 5 bytes

return (dht[4] == (dht[0] + dht[1] + dht[2] + dht[3])); // Verify checksum
```

⚠️ Uses **bit-banging** and precise delay-based timing to implement the one-wire DHT11 protocol.

6️⃣ ADC Setup for LDR

void ADC_Manual_Init()

```
RCC_APB2ENR |= (1 << 2);  // Enable GPIOA
RCC_APB2ENR |= (1 << 9);  // Enable ADC1
GPIOA_CRL &= ~(0xF << 16); // Clear bits for PA4
GPIOA_CRL |=  (0x0 << 16); // Analog mode
ADC1_CR2 |= (1 << 0);     // Turn on ADC
ADC1_CR2 |= (1 << 1);     // Enable continuous conversion
ADC1_CR2 |= (1 << 20);    // Enable external trigger
```

unsigned int ADC_Read_PA4()

```
ADC1_SQR3 = 4;              // Select Channel 4 (PA4)
ADC1_CR2 |= (1 << 22);        // Start conversion
while (!(ADC1_SR & (1 << 1)));  // Wait for EOC
return ADC1_DR;             // Return result
```

Reads analog light level from PA4 using 12-bit ADC.

7️⃣ Main Program Logic

```
void main() {
  // LCD and GPIO initialization
  GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_1 | _GPIO_PINMASK_2);
  GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_0 | _GPIO_PINMASK_1 |
                      _GPIO_PINMASK_10 | _GPIO_PINMASK_11);
  ADC_Manual_Init();
  Lcd_Init();
  Delay_ms(100);
  Lcd_Cmd(_LCD_CLEAR);
  Lcd_Cmd(_LCD_CURSOR_OFF);
```

```
while(1) {
  Lcd_Cmd(_LCD_CLEAR);

  // Read from DHT11
  if(DHT11_Read()) {
    Lcd_Out(1, 1, "T:");
    Lcd_Chr(1, 3, dht[2]/10 + '0');
    Lcd_Chr(1, 4, dht[2]%10 + '0');
    Lcd_Chr(1, 5, 223);  // Degree symbol
    Lcd_Chr(1, 6, 'C');

    Lcd_Out(1, 8, "H:");
    Lcd_Chr(1,10, dht[0]/10 + '0');
    Lcd_Chr(1,11, dht[0]%10 + '0');
    Lcd_Chr(1,12, '%');
  } else {
    Lcd_Out(1, 1, "DHT11 Error");
  }

  // Read from LDR (ADC)
  light = ADC_Read_PA4();
  WordToStr(light, text);
  Lcd_Out(2, 1, "Light:");
  Lcd_Out(2, 8, text);

  Delay_ms(2000);  // Update every 2 seconds
```

```
  }
}
```

🧠 **Key Output Example:**

LCD Line 1	LCD Line 2
T:23°C H:45%	Light: 1267

📊 **What You Learn in This Project:**

Concept	Usage
DHT11 bit-banging	Interfacing non-standard digital sensor
ADC (Analog)	Reading environmental data (LDR) via voltage divider
LCD (4-bit mode)	Efficient UI for data display
Microsecond delays	Required for DHT11 protocol timing
Checksum verification	Ensures valid DHT11 data

⚙️ **Potential Enhancements:**

- Add **EEPROM logging** of sensor values
- Transmit data via **USART/Bluetooth** to a phone or PC
- Add **alert system** if temperature > threshold (like fan or buzzer)
- Use **I2C LCD** to reduce pin count

Would you like:

- Proteus circuit diagram setup?
- STM32CubeMX version of the same project?
- Move forward to **Project 11: Traffic Light System using Timer & GPIO**?

Great! Let's now explain **Project 11: Traffic Light System using Timer and GPIO**, with **interrupt-based traffic light control** in MikroC for STM32.

🚦 Project 11: Traffic Light Control with External Interrupt

🎯 Objective:

Build a basic traffic light system using:

- **Three LEDs (Red, Yellow, Green)**
- **EXTI (external interrupt) on PC13** to switch states
- Use of **GPIO**, **interrupt vector**, and **state machine**

🧩 Hardware Summary

Component	Purpose
STM32F103C6	Main microcontroller
LEDs (R, Y, G)	Represent traffic lights (PA0–PA2)
Push button (PC13)	Trigger interrupt to change light
Proteus	For simulation

📜 Full Code Explanation

🔧 GPIO Definitions

```
sbit RED_LED at GPIOA_ODR.B0;
sbit YELLOW_LED at GPIOA_ODR.B1;
sbit GREEN_LED at GPIOA_ODR.B2;
```

- Define names for each LED connected to PA0, PA1, and PA2

🔢 State Machine Variable

unsigned int traffic_light_state;

- Will hold current light state (0 = Red, 1 = Yellow, 2 = Green)

⏱️ Interrupt Handler for PC13 (EXTI13)

void EXTI15_10_IRQHandler() iv IVT_INT_EXTI15_10 {

if (EXTI_PR.B13) {

EXTI_PR.B13 = 1; // Clear interrupt flag

📝 When PC13 is triggered:

- Clear the flag
- Reset all LEDs
- Check traffic_light_state and turn ON the correct LED

🔁 LED State Logic

RED_LED = 0; YELLOW_LED = 0; GREEN_LED = 0;

switch (traffic_light_state) {

case 0: RED_LED = 1; break;

case 1: YELLOW_LED = 1; break;

case 2: GREEN_LED = 1; break;

}

Then increment and wrap around:

traffic_light_state = (traffic_light_state + 1) % 3;

Cycle: Red → Yellow → Green → Red...

⚙️ GPIO Setup

```
void setup_gpio() {
  RCC_APB2ENR |= (1 << 2); // GPIOA
  RCC_APB2ENR |= (1 << 4); // GPIOC

  GPIOA_CRL |= 0x00000022; // PA0–PA2 as Output Push-Pull

  GPIOC_CRH &= ~(0xF << 20);    // Clear CNF13 & MODE13
  GPIOC_CRH |=  (0x4 << 20);    // Input Floating (CNF=01, MODE=00)
}
```

🔁 EXTI (External Interrupt) Setup

```
void setup_exti() {
  RCC_APB2ENR |= (1 << 0);    // Enable AFIO

  AFIO_EXTICR4 &= ~(0x000F << 4); // Select Port C for EXTI13
  AFIO_EXTICR4 |=  (0x0002 << 4); // PC13 = EXTI13

  EXTI_IMR  |= (1 << 13); // Unmask EXTI13
  EXTI_RTSR |= (1 << 13); // Rising edge trigger

  NVIC_IntEnable(IVT_INT_EXTI15_10); // Enable EXTI in NVIC
}
```

🧠 main() Routine

```
void main() {
```

```
    setup_gpio();    // Configure GPIOs
    setup_exti();    // Configure External Interrupt

    // Start with Red light
    RED_LED = 1;
    YELLOW_LED = 0;
    GREEN_LED = 0;
    traffic_light_state = 0;

    EnableInterrupts(); // Enable global interrupt (PRIMASK)

    while (1) {
      // Nothing here – ISR does the job
    }
}
```

🧪 **Proteus Simulation Overview**

🧩 **Schematic Highlights**

Pin	Connected to
PA0	Red LED
PA1	Yellow LED
PA2	Green LED
PC13	Push button input
8 MHz	External crystal

- PC13 is pulled up and pulled LOW when button pressed → **rising edge interrupt** triggers state change

---

✅ Project Behavior

Press Button	Light Turns ON	State
1st Press	RED	0
2nd Press	YELLOW	1
3rd Press	GREEN	2
4th Press	RED	0

- ✔ Clean transitions
- ✔ Non-blocking (main loop free)
- ✔ Efficient via **interrupts**

📚 What You Learn

Concept	Description
GPIO Input/Output	Configure LED and button pins
Interrupts with EXTI	React immediately to button press
NVIC + ISR Structure	Use STM32 vector table and flags
State Machine Design	Build cyclic LED control logic

🚀 Optional Enhancements

- Add **LCD** to display “STOP”, “WAIT”, “GO”
- Use **Timer interrupt** for timed light change
- Simulate **pedestrian button** with different EXTI
- Add **buzzer** for alert when switching states

Would you like to:

- Add timing with SysTick or Timer2?
- Move to the next project?
- Or simulate emergency override input?

Let me know!

Excellent! Let's now explain **Project 12: Fan Speed Controller using ADC (Temperature-based Control)** in MikroC with STM32.

🌡️ Project 12: Automatic Fan Control using TMP36 Sensor

🎯 Objective:

Use **TMP36 analog temperature sensor** to:

- Measure **ambient temperature**
- Automatically **activate a fan** via relay when temp > **75°C**
- Show status using **LED**
- Use STM32 ADC to convert analog voltage to temperature

🛠️ Hardware Components:

Component	Description
STM32F103C6	Main controller
TMP36	Temperature sensor (analog output)
Relay Module	Switches fan ON/OFF
Fan	Controlled via relay
LED	Indicates fan activity
Resistors	Current limiting for LED and transistor
Proteus Sim	Used for simulation of the circuit

🧠 TMP36 Temperature Sensor Overview

- Output = 0.5 V + (Temp × 10 mV)
- So:

- 25°C → 0.75 V
- 75°C → 1.25 V

- Formula:

Temperature (°C)=(VOUT−0.5)×100\text{Temperature (°C)} = (V_{OUT} - 0.5) \times 100

📜 **Code Breakdown**

🔹 **Macro Definitions**

```
#define LED    GPIOB_ODR.B1   // PB1 - LED
#define MOTOR  GPIOB_ODR.B2   // PB2 - Relay
```

🔹 **Main Routine**

```
void main() {
  unsigned int adc_value;
  float voltage, temperature;

  Init_All();      // Initialize peripherals
  LED = 0;
  MOTOR = 0;

  while (1) {
    adc_value = read_ADC(); // Read analog value from TMP36
```

🔹 **Analog to Voltage**

```
voltage = (adc_value * 3.3) / 4095.0;
```

- Converts ADC value (0–4095) to voltage (0–3.3V)

🔹 **Voltage to Temperature**

```
temperature = (voltage - 0.5) * 100.0;
```

- Applies TMP36 formula
  e.g. 1.25V - 0.5 = 0.75 × 100 = 75°C

🔹 **Control Logic**

```
if (temperature > 75.0) {
  LED = 1;
  MOTOR = 1;  // Turn fan ON
} else {
  LED = 0;
  MOTOR = 0;  // Turn fan OFF
}
```

- Turns on fan and LED if temp exceeds threshold

🔹 **Delay**

```
Delay_ms(300); // Avoid flickering due to rapid sensor changes
```

⚙️ **Init_All() – System Initialization**

```
void Init_All() {
  // Enable clocks
  RCC_APB2ENR |= (1 << 2); // GPIOA
  RCC_APB2ENR |= (1 << 3); // GPIOB
```

- Enables GPIOA and GPIOB

🔹 **PB1 & PB2 Output (LED, Relay)**

```
GPIOB_CRL &= ~(0xFF << 4); // Clear PB1 & PB2 config
GPIOB_CRL |=  (0x22 << 4); // Push-pull output, 2 MHz
```

🔹 **PA0 as Analog Input (TMP36)**

```
GPIOA_CRL &= ~(0xF << 0);
GPIOA_CRL |=  (0x0 << 0); // Analog mode
```

🔹 **ADC Configuration**

```
RCC_APB2ENR |= (1 << 9);   // ADC1 clock enable
ADC1_CR2 = 0;
ADC1_CR1 = 0;
ADC1_SMPR2 |= (7 << 0);    // Max sample time
ADC1_CR2 |= (1 << 0);      // ADC ON
Delay_ms(1);
```

🔍 **read_ADC() – ADC Sampling**

```
unsigned int read_ADC() {
  ADC1_CR2 |= (1 << 22);      // Start conversion
  while (!(ADC1_SR & (1 << 1))); // Wait for EOC
  return ADC1_DR;
}
```

🧪 **Proteus Simulation Notes**

- **TMP36** connected to PA0

- Output voltage visualized
- **Relay** connected to PB2 to activate fan
- **LED (D4)** shows fan status
- Uses external 12V supply for fan & relay

✅ Fan activates above 75°C
✅ LED gives status indication
✅ TMP36 reads analog temperature

✅ What You Learn

Concept	Explanation
ADC Configuration	Convert analog temp to digital
GPIO Output	Drive relay and LED
Temperature Conversion	Use of TMP36 linear formula
Conditional Control	IF statements to automate based on input

🚀 Extension Ideas

- Use **LCD** to show live temperature
- Replace with **PWM fan** for speed control
- Add **EEPROM** to save max temp recorded
- Set temperature threshold via **buttons or potentiometer**

Would you like to:

- Simulate this with LCD temp display?
- Replace relay with **PWM fan** control?
- Or move to **Project 13**?

Perfect! Let's now explain **Project 13: Motion Detector using PIR Sensor** using STM32 and MikroC.

🚶 Project 13: PIR-based Motion Detection System

🎯 Objective:

- Detect motion using a **PIR sensor** connected to **PA0**
- Trigger a **buzzer (on PB0)** for 1 second when motion is detected
- Use **external interrupt (EXTI0)** for instant motion response

---

🛠️ Hardware Components:

Component	Purpose
PIR Sensor	Detects motion
STM32F103C6	Processes input and controls buzzer
Buzzer	Audible alarm when motion is detected
Crystal	8 MHz oscillator

🧠 How PIR Sensor Works

- Detects infrared radiation changes (motion)
- Output: HIGH (logic 1) when motion is detected, else LOW
- Connected to **PA0**, which triggers an **external interrupt (EXTI0)**

📜 Code Explanation

🔸 Pin and Flag Declaration

```
sbit BUZZER at GPIOB_ODR.B0;    // PB0 controls the buzzer
volatile unsigned char motion_detected = 0;
```

- Buzzer is connected to **PB0**
- motion_detected flag is set inside the ISR when motion occurs

---

⚡ Interrupt Service Routine – EXTI0

```
void EXTI0_ISR() iv IVT_INT_EXTI0 ics ICS_AUTO {
  EXTI_PR.B0 = 1;          // Clear interrupt pending bit
  motion_detected = 1;     // Set motion flag
}
```

- Executed when PA0 detects a **rising edge**
- Clears interrupt and sets a flag so main loop can act

🔧 GPIO Configuration

```
GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_0);  // PB0 for buzzer
GPIO_Digital_Input(&GPIOA_BASE, _GPIO_PINMASK_0);   // PA0 for PIR
```

⚙️ EXTI Setup for PA0

```
RCC_APB2ENR |= (1 << 0);        // Enable AFIO clock
AFIO_EXTICR1 = 0x0000;          // Map EXTI0 to Port A
EXTI_IMR.B0 = 1;                // Unmask interrupt line 0
EXTI_RTSR.B0 = 1;               // Enable rising edge trigger
NVIC_IntEnable(IVT_INT_EXTI0);  // Enable EXTI0 in NVIC
```

- Enables interrupt on **PA0**
- Responds to **rising edge** when PIR detects motion

🏁 Main Function Logic

```
void main() {
  // Setup
  GPIO & EXTI config
  EnableInterrupts();  // Global interrupt enable
  BUZZER = 0;

  while (1) {
    if(motion_detected) {
      BUZZER = 1;        // Activate buzzer
      Delay_ms(1000);     // Wait 1 second
      BUZZER = 0;        // Turn off
      motion_detected = 0;  // Reset flag
    }
  }
}
```

✅ Detects motion
✅ Turns buzzer ON for 1s
✅ Waits for next detection

💉 Proteus Simulation Summary

🧩 Circuit Flow:

- **PIR Sensor** VCC → 5V, GND → GND, OUT → PA0
- **Buzzer** → PB0 via resistor
- **EXTI0** triggered on rising edge when PIR sees motion
- **LED (optional)** can be added for debugging

✅ What You Learn from This Project

Concept	Explanation
External Interrupt (EXTI0)	Fast hardware-level response on GPIO
ISR usage	Efficient and non-blocking execution
PIR Interfacing	Practical application of motion detection
Buzzer control	Simple GPIO output for alerts

🚀 Extension Ideas

- Add **LCD or UART** to display "Motion Detected!"
- Count number of detections
- Add **night-only mode** using LDR + ADC
- Add **recording or alarm system**

Would you like to:

- Simulate a real-world delay-off timer?
- Add motion logging to EEPROM?
- Move to Project 14?

Let me know!

Excellent! Let's now explain **Project 14: Gas Leakage Detection using ADC** with STM32 and MikroC in full detail.

🧪 Project 14: Gas Sensor (Analog) with LED Alert System

🎯 Objective:

- Use an analog **gas sensor (e.g. MQ-2)** connected to **PA0** (ADC input)
- Continuously monitor gas concentration
- Trigger **LED (on PB1)** if ADC value exceeds a threshold (800)

🛠️ Hardware Summary

Component	Purpose
STM32F103C6	Main microcontroller
Gas sensor	Analog output proportional to gas
LED (D4)	Turns ON when gas threshold exceeded
Potentiometer	Simulates analog gas sensor in Proteus
10kΩ resistor	Pull-down / series resistor for RV1
Crystal	8 MHz external clock

📜 MikroC Code Explanation

🔸 Pin Definitions

```
#define Gas_Sensor GPIOA_ODR.B0
#define LED        GPIOB_ODR.B1
```

- **Gas_Sensor**: input via PA0 (analog pin)
- **LED**: output via PB1 (alert indicator)

🔧 GPIO Configuration

```
GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_1, _GPIO_CFG_MODE_OUTPUT | _GPIO_CFG_SPEED_MAX | _GPIO_CFG_OTYPE_PP);

GPIO_Config(&GPIOA_BASE, _GPIO_PINMASK_0, _GPIO_CFG_MODE_ANALOG);
```

- PB1 configured as digital output for LED
- PA0 configured as analog input (ADC input for gas sensor)

🧪 ADC Initialization

```
void ADC1_Init() {
  RCC_APB2ENR |= (1 << 9);  // Enable ADC1 clock
  ADC1_CR2 = 0;
  ADC1_CR1 = 0;
  ADC1_SMPR2 |= (7 << 0);   // Max sampling time
  ADC1_CR2 |= (1 << 0);     // ADC ON
  Delay_ms(1);
}
```

- Enables ADC1 with long sampling for accuracy

📩 Reading the ADC

```
unsigned int read_ADC() {
  ADC1_CR2 |= (1 << 22);        // Start conversion
  while (!(ADC1_SR & (1 << 1)));  // Wait for EOC
  return ADC1_DR;               // Read ADC data
```

```
}
```

- Starts conversion and returns 12-bit ADC value (0–4095)

🏁 Main Application Logic

```
void main() {
  unsigned int Gas_Value = 0;
  float GAS;

  // GPIO & ADC setup
  GPIO_Config(...);
  ADC1_Init();

  Gas_Sensor = 0;

  while(1) {
    Gas_Value = read_ADC();
    GAS = Gas_Value;

    if(GAS > 800) {
      LED = 1; // Gas detected → alert
    } else {
      LED = 0;
    }
  }
}
```

📊 Threshold Explanation

- GAS > 800 means dangerous gas level detected
- You can calibrate this threshold based on the **actual sensor output** for the target gas

---

💉 Proteus Simulation Description

Element	Connected to
RV1	Simulates gas level (analog voltage to PA0)
PA0	ADC input (gas sensor)
PB1	LED (D4) alert output
R4, R8	Current limiting for LED
Crystal + C1, C2	8MHz system clock

🧩 Simulation Behavior:

- **Increase RV1** ⇒ simulates increased gas
- Once analog voltage > threshold, ADC > 800
- LED (D4) turns ON to indicate gas detection

✅ What You Learn from This Project

Concept	Explanation
ADC in STM32	Read analog voltage and convert to digital
GPIO output control	Drive alert devices (LED, buzzer)
Real-world sensor input	Simulate and threshold analog signals
Safety application logic	Create critical-condition response system

---

🚀 Extension Ideas

- Add **buzzer** alert along with LED
- Show **live ADC value or ppm** on **LCD**
- Send gas alert via **UART or GSM**
- Add **EEPROM logging** of gas events

Would you like to:

- Simulate this with LCD reading ppm values?
- Move to Project 15?
- Add buzzer or GSM message alert?

Let me know how you want to proceed!

Great! Let's now explain **Project 16: Fire Alarm System using LM35 Temperature Sensor** with STM32 and MikroC in full detail.

🔥 Project 16: Temperature-based Fire Alarm using LM35

🎯 Objective:

- Use an **LM35 analog temperature sensor** to measure room temperature
- Convert analog signal to temperature via **ADC**
- If temperature **exceeds 30°C**, trigger a **buzzer alarm**

🛠️ Hardware Components

Component	Role
STM32F103C6	Reads temperature & controls buzzer
LM35	Analog temperature sensor (0.01V/°C)
Buzzer	Activates when fire is detected
Crystal	8 MHz system clock

📜 MikroC Code Explanation

🔧 Macro & Constant Definitions

```
sbit BUZZER at GPIOB_ODR.B0;  // Buzzer connected to PB0


#define TEMP_THRESHOLD 30.0   // Fire alarm threshold in °C
#define VREF 3.3              // ADC reference voltage (3.3V)
```

#define ADC_RESOLUTION 4096.0 // 12-bit ADC (0–4095)

🧠 Function: readTemperature()

🕹️ ADC Read & Conversion Logic

unsigned int adc_value;

float voltage, temperature;

adc_value = ADC1_Read(0); // Read from channel 0 (PA0)

- LM35 output (in volts) is read using ADC → value 0–4095

⚡ Convert ADC Value to Voltage

voltage = (adc_value * VREF) / ADC_RESOLUTION;

- Example: adc = 620 → voltage = (620 × 3.3) / 4096 ≈ 0.5 V

🌡️ Convert Voltage to Temperature

temperature = voltage * 100.0;

- LM35 outputs 10mV per °C → 0.5V = 50°C

✅ Return Temperature

return temperature;

🏁 Main Function Flow

🔧 Setup

GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_0); // PB0 = buzzer

ADC1_Init();   // Init ADC

ADC_Set_Input_Channel(_ADC_CHANNEL_0);   // Use PA0

```
BUZZER = 0;                    // Start OFF
```

🔁 Main Loop

```
while (1) {
  float temperature = readTemperature();   // Get temp

  if (temperature > TEMP_THRESHOLD) {
    BUZZER = 1;  // 🔔 Alarm if temp > 30°C
  } else {
    BUZZER = 0;  // 🔕 No alarm if normal
  }

  Delay_ms(500);  // Stability / debounce
}
```

💉 Proteus Simulation Setup

Connection	Details
LM35 Output (Vout)	Connected to **PA0** (ADC channel 0)
Buzzer	Connected to **PB0**
Power	LM35 powered by 5V, outputs 10mV/°C
Crystal + Caps	8 MHz system oscillator

🔁 Behavior

LM35 Temp (°C)	Output Voltage (V)	ADC Reading (approx)	Buzzer
25	0.25 V	~310	OFF
30	0.30 V	~372	OFF
>30	>0.30 V	>372	ON

✅ What You Learn

Concept	Explanation
ADC Conversion	Map analog voltage to digital
LM35 Temperature Interfacing	Understand sensor scaling (10 mV/°C)
Threshold Detection Logic	Fire detection based on reading
GPIO Output Control	Drive buzzer on condition

🚀 Enhancement Ideas

- Show temperature on **LCD or Serial Monitor**
- Replace buzzer with **fan/relay** to activate cooling
- Log temperature over time to **EEPROM**
- Allow **dynamic threshold** via keypad or potentiometer

Would you like to:

- Add LCD for temperature display?
- Simulate a fan-based temperature controller?
- Move to **Project 17**?

Just tell me!

Great! Let's now fully explore **Project 17: Automatic Fan Control Based on Light Intensity (using LDR + ADC)**.

🌞 Project 17: LDR-Based Fan (or Light) Control with STM32

🎯 Objective:

Automatically **turn ON/OFF a fan or light** based on **ambient light** detected by an **LDR** (Light Dependent Resistor).

🛠️ Hardware Required:

Component	Function
LDR + Resistor	Voltage divider → connected to ADC
STM32F103C6/C8	Microcontroller
Relay or Fan	Device to control (connected to PA1)
16x2 LCD	Display light level (optional)

📜 Full Code with Line-by-Line Comments

```
char txt[16];          // LCD buffer
unsigned int adc_value;   // ADC raw value from LDR
float light_level;      // Interpreted light level


void main() {
  // LCD pin definitions
  sbit LCD_RS at GPIOB_ODR.B0;
  sbit LCD_EN at GPIOB_ODR.B1;
  sbit LCD_D4 at GPIOB_ODR.B2;
```

```
  sbit LCD_D5 at GPIOB_ODR.B10;
  sbit LCD_D6 at GPIOB_ODR.B11;
  sbit LCD_D7 at GPIOB_ODR.B12;

  GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_1); // PA1 output to control relay/light
  ADC1_Init();  // Initialize ADC peripheral
  Lcd_Init();   // Initialize LCD
  Lcd_Cmd(_LCD_CLEAR);
  Lcd_Cmd(_LCD_CURSOR_OFF);
  Lcd_Out(1, 1, "LDR Control");

  while (1) {
    adc_value = ADC1_Get_Sample(0);  // Read analog value from channel 0 (PA0)

    // Convert ADC to voltage level (for readability; optional)
    light_level = (adc_value * 3.3) / 4096.0; // Converts to voltage

    // Show voltage on LCD
    FloatToStr(light_level, txt);
    Lcd_Out(2, 1, txt);
    Lcd_Out_CP(" V");  // V for volts

    // If bright light (voltage > 2V), turn ON the fan
    if (light_level > 2.0) {
      GPIOA_ODR.B1 = 1;  // Turn ON fan or light
```

```
    }
    else {
      GPIOA_ODR.B1 = 0;  // Turn OFF fan or light
    }

    Delay_ms(1000);  // Update every 1 second
  }
}
```

📘 Detailed Explanation of Each Section

🔹 **char txt[16];**

- Buffer to store the voltage as a string for LCD display.

🔹 **unsigned int adc_value;**

- Holds the raw 12-bit ADC result from **channel 0 (PA0)**, where the LDR is connected.

🔹 **float light_level;**

- Converts the raw ADC result to **voltage** (0–3.3V), which correlates with **light intensity**.

🔹 **sbit LCD_RS ... LCD_D7**

- Connect the LCD in **4-bit mode** using specific **Port B pins**.
- These are needed for controlling and displaying information on the LCD.

🔹 **GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_1);**

- Configures **PA1** as a digital output pin to control a device (fan/relay/light).

- **ADC1_Init();**
  - Initializes **ADC1** peripheral to read analog signals.

- **Lcd_Init(); ...**
  - Initializes the LCD and shows “LDR Control” as a welcome message on line 1.

- **while (1) {**
  - Infinite loop to continuously read LDR and control output.

- **adc_value = ADC1_Get_Sample(0);**
  - Reads analog signal from **PA0** where LDR is connected in a **voltage divider** setup.

- **light_level = (adc_value * 3.3) / 4096.0;**
  - Converts the 12-bit ADC value (0–4095) into a **voltage** (0–3.3V)
  - The higher the voltage → the **brighter** the light

- **FloatToStr(light_level, txt);**
  - Converts float voltage to a string for LCD output

- **Lcd_Out(2, 1, txt);**
  - Displays voltage value starting from line 2, column 1

- **Lcd_Out_CP(" V");**

- Appends the **“ V” unit** next to the value

🔹 **if (light_level > 2.0) { ... }**

- If it’s **bright** enough (e.g., direct sunlight or room light), the voltage goes above 2.0V
- **Turn ON fan/light** by setting PA1 HIGH

🔹 **else { GPIOA_ODR.B1 = 0; }**

- If light level is low, turn OFF the fan/light (PA1 LOW)

🔹 **Delay_ms(1000);**

- Delay to avoid flickering and excessive updating

🧠 **LDR Working Principle Recap**

- LDR + Resistor form a **voltage divider**:

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

  - When light increases, LDR resistance drops → voltage at ADC pin increases.
  - When dark, LDR resistance increases → voltage at ADC pin drops.

✅ **Concepts Covered**

Concept	Explanation
LDR & ADC Integration	Read light levels using analog input
Voltage Divider Logic	Understanding LDR behavior with fixed resistor
GPIO Output	Turn on/off devices based on light threshold

Concept	Explanation
LCD Integration	Display live voltage or light level

🔧 Possible Extensions

- Add a **buzzer** for light/dark alert
- Use **PWM** to control fan speed based on light level
- Display brightness level (e.g., LOW/MEDIUM/HIGH) instead of voltage
- Implement **logging** of brightness levels using EEPROM or UART

Would you like:

- To simulate this with Proteus?
- Or proceed to additional project ideas?
- Or revisit any specific technical topic in these projects?

Let me know how you'd like to proceed!

Excellent — you've now shared the **actual C code** used for **Project 18: DC Motor with LCD + PWM + LDR Sensor**, implemented using **direct register-level programming** for STM32 (not library functions like MikroC).

Let's now **break it down in full detail**: line by line, register by register, and function by function.

🧠 Overview of the Project

- Use **ADC** to read the LDR voltage from **PA0**.
- Use **PWM on TIM2 Channel 2 (PA1)** to control **motor speed** based on light intensity.
- Directly manipulate **RCC, GPIO, TIM2, and ADC1 registers**.

🔷 Section 1: Register Definitions

These lines define base addresses for peripheral registers:

#define RCC_BASE 0x40021000

#define GPIOA_BASE 0x40010800

#define TIM2_BASE 0x40000000

#define ADC1_BASE 0x40012400

👉 These are **memory-mapped base addresses** of key peripherals.

Next, specific **register offsets** are defined as volatile pointers. For example:

#define RCC_APB2ENR (*(volatile unsigned long *)(RCC_BASE + 0x18))

#define GPIOA_CRL (*(volatile unsigned long *)(GPIOA_BASE + 0x00))

These allow direct access to registers like:

- **RCC_APB2ENR**: Clock enable for GPIO and ADC
- **GPIOA_CRL**: Configure pin modes for PA0–PA7
- **TIM2 registers**: Set up Timer 2 for PWM
- **ADC1 registers**: Configure ADC sampling and conversion

🔷 **Section 2: main() Function**

```
int main(void) {
  unsigned int adc_value;
  unsigned int duty;
  ADC1_Init();       // Initialize ADC (LDR)
  TIM2_PWM_Init();    // Initialize PWM for motor (PA1)

  while(1) {
    adc_value = ADC1_Read();  // Read LDR analog input
    duty = (adc_value * TIM2_ARR) / 4095; // Scale to PWM range
    Set_PWM_Duty(duty);     // Update PWM duty
    delay(50000);          // Small delay
  }

  return 0;
}
```

🔍 **Line-by-Line:**

- ADC1_Init();: Set up ADC1 and configure PA0 as analog input.
- TIM2_PWM_Init();: Set up Timer 2 Channel 2 as PWM output on PA1.
- adc_value = ADC1_Read();: Get light intensity (0–4095).
- duty = (adc_value * TIM2_ARR) / 4095;: Scale to PWM duty (0–999).
- Set_PWM_Duty(duty);: Adjust PWM based on brightness.
- delay(50000);: Simple software delay loop.

🔷 **Section 3: delay()**

```
void delay(volatile unsigned int count) {
```

```
  while(count--) { ; }
}
```

- Basic software delay loop (blocking).
- Used to slow down ADC and PWM updates slightly.

🔷 Section 4: ADC1_Init()

```
void ADC1_Init(void) {
  RCC_APB2ENR |= (1 << 2);  // Enable GPIOA clock
  RCC_APB2ENR |= (1 << 9);  // Enable ADC1 clock

  GPIOA_CRL &= ~(0xF << 0); // Clear PA0 bits (Analog mode: MODE=00, CNF=00)

  GPIOA_CRL &= ~(0xF << 4); // Clear PA1 bits
  GPIOA_CRL |= (0xA << 4);  // PA1: Alternate Function Output Push-Pull (CNF=10, MODE=10)

  ADC1_CR2 = 0;          // Reset CR2
  ADC1_CR1 = 0;          // Reset CR1
  ADC1_SMPR2 |= (7 << 0);   // Maximum sample time for channel 0 (239.5 cycles)
  ADC1_SQR1 = 0;          // 1 conversion
  ADC1_SQR3 = 0;          // First channel = 0 (PA0)
  ADC1_CR2 |= (1 << 0);     // ADON: Turn on ADC
  delay(1000);           // Stabilization time
}
```

🔍 Key Actions:

- Enable GPIOA and ADC1 clocks.

- Configure:
    - **PA0** as analog input (for LDR)
    - **PA1** as alternate function output (for PWM)
- Set **sampling time** for accuracy.
- Set ADC to perform **1 conversion** on **channel 0 (PA0)**.
- Turn on ADC with **ADON bit**.

🔷 Section 5: ADC1_Read()

```
unsigned int ADC1_Read(void) {
  ADC1_SQR3 = 0;         // Select channel 0 (PA0)
  ADC1_CR2 |= (1 << 22);   // Start conversion (SWSTART)
  while(!(ADC1_SR & (1 << 1))); // Wait for EOC (End of Conversion)
  return ADC1_DR;        // Return digital result
}
```

🔍 Step-by-Step:

- Set sequence to channel 0 (PA0).
- Start ADC conversion.
- Wait for **EOC (End Of Conversion)** bit.
- Read result from **ADC1_DR** (Data Register).

🔷 Section 6: TIM2_PWM_Init()

```
void TIM2_PWM_Init(void) {
  RCC_APB1ENR |= (1 << 0);   // Enable TIM2 clock
  TIM2_CR1 = 0;          // Reset TIM2

  TIM2_PSC = 7;          // Prescaler: 8 MHz / (7+1) = 1 MHz
```

```
  TIM2_ARR = 999;          // Auto-reload: PWM period = 1 kHz

  // Set PWM Mode 1 on Channel 2, preload enabled
  TIM2_CCMR1 &= ~(0xFF << 8);
  TIM2_CCMR1 |= (0x6 << 12) | (1 << 11);      // OC2M = 110, OC2PE = 1

  TIM2_CCER |= (1 << 4);     // Enable CH2 output (CC2E = 1)
  TIM2_CR1 |= (1 << 7);      // ARPE enable (Auto-reload preload)
  TIM2_EGR |= 1;           // Generate update event
  TIM2_CR1 |= 1;           // Enable timer
}
```

🔍 **Explanation:**

- Timer 2 runs at 1 MHz and generates PWM at 1 kHz.
- **Channel 2** is configured in **PWM Mode 1** (active as long as CNT < CCR2).
- Output is enabled on **PA1**.
- Timer is started by setting **CEN (bit 0)**.

🔷 **Section 7: Set_PWM_Duty()**

```
void Set_PWM_Duty(unsigned int duty) {
  TIM2_CCR2 = duty;  // Set duty cycle for channel 2
}
```

- Updates **CCR2**, which controls the duty cycle of **PWM output on PA1**.

✅ **Final Summary:**

Module	Purpose
ADC1	Reads LDR voltage from PA0
TIM2_CH2	Generates PWM on PA1
PWM Duty	Proportional to light level
GPIO	PA0 = Analog In, PA1 = PWM Out

This is a **clean, low-level embedded system project** that teaches:

- Peripheral register-level programming
- ADC configuration
- Timer/PWM setup
- Hardware interfacing basics

Would you like me to:

- Provide a Proteus simulation diagram for this circuit?
- Explain how to extend this project with an LCD or interrupt?
- Move to **Project 19**?

Let me know what you'd like next!

Let's now explain **Project 20: Energy Meter using ADC** — it measures **current using ACS712** and displays it on a **16x2 LCD** using an STM32F103 microcontroller and MikroC.

⚡ Project Goal

Measure **current** using an **ACS712** sensor connected to **ADC (PA0)**, process the analog voltage, and display the result on **LCD** in real-time.

🧩 Hardware Summary

Component	Function
STM32F103C6	Main microcontroller
ACS712	Current sensor (0 A = 2.5V)
16x2 LCD	Display the current
RV1	Contrast for LCD
VOUT → PA0	Analog current signal to ADC

📜 Code Breakdown (MikroC)

🔸 Pin Definitions

```
#define RS GPIOA_ODR.B1
#define RW GPIOA_ODR.B2
#define E  GPIOA_ODR.B3
#define LCD_DATA GPIOB_ODR
```

- RS: Register Select
- RW: Read/Write
- E: Enable

- LCD_DATA: All 8 bits of LCD data lines go to **PORTB**

🔧 main() – System Setup and Loop

📍 Variables

unsigned int adc_raw;

float adc_voltage, curr_amp;

char curr_str[10];

- adc_raw: 12-bit ADC result (0–4095)
- adc_voltage: voltage output from ACS712
- curr_amp: calculated current in Amperes
- curr_str: ASCII string for LCD

📍 GPIO & LCD Initialization

GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_ALL, _GPIO_CFG_MODE_OUTPUT | _GPIO_CFG_SPEED_MAX | _GPIO_CFG_OTYPE_PP);

GPIO_Config(&GPIOA_BASE, _GPIO_PINMASK_1 | _GPIO_PINMASK_2 | _GPIO_PINMASK_3, _GPIO_CFG_MODE_OUTPUT | _GPIO_CFG_SPEED_MAX | _GPIO_CFG_OTYPE_PP);

GPIO_Config(&GPIOA_BASE, _GPIO_PINMASK_0, _GPIO_CFG_MODE_ANALOG); // PA0 = analog input

- PortB: Output (LCD data)
- PA1–PA3: Control lines for LCD
- PA0: Analog input from **ACS712**

📍 LCD Setup

lcd_Send_command(0x38); // 8-bit, 2 lines

lcd_Send_command(0x0C); // Display ON

lcd_Send_command(0x01); // Clear display

lcd_Send_command(0x06); // Entry mode: auto increment

🔁 Loop for Reading and Displaying Current

📍 ADC Reading and Conversion

adc_raw = read_ADC();

adc_voltage = (adc_raw * 3.3) / 4095.0;

curr_amp = (adc_voltage - 2.5) / 0.1;

- ADC reads the analog voltage from ACS712.
- ACS712 center = 2.5V → 0 A
- 100 mV per Amp: so (V - 2.5)/0.1 = current

📍 Absolute Value (no sign)

if (curr_amp < 0) curr_amp = -curr_amp;

📍 Display Current on LCD

int_to_string((unsigned int)curr_amp, curr_str); // Convert number to string

lcd_Send_command(0x01); // Clear LCD

lcd_display_string("Current: ");

lcd_display_string(curr_str);

lcd_display_string(" A");

Delay_ms(1000);

💉 ADC Read Function

```
unsigned int read_ADC() {
  ADC1_CR2 |= (1 << 22);       // Start conversion
  while (!(ADC1_SR & (1 << 1))); // Wait for EOC
```

```
  return ADC1_DR;
}
```

- Starts ADC1 conversion
- Waits for **End Of Conversion**
- Returns digital value

📺 LCD Control Functions

🔹 Send Command

```
void lcd_Send_command(unsigned char command) {
  RS = 0; RW = 0; E = 1;
  LCD_DATA = command;
  E = 0; Delay_ms(50);
}
```

🔹 Display Character

```
void lcd_display(unsigned char Data) {
  RS = 1; RW = 0; E = 1;
  LCD_DATA = Data;
  E = 0; Delay_ms(50);
}
```

🔹 Display String

```
void lcd_display_string(char* str) {
  while (*str) lcd_display(*str++);
}
```

🔠 Integer to String (No sprintf)

```
void int_to_string(unsigned int value, char* str) {
  if (value == 0) { str[0] = '0'; str[1] = '\0'; return; }

  unsigned int i = 0, j;
  char curr_str[10];

  while (value != 0) {
    curr_str[++i] = (value % 10) + '0';
    value /= 10;
  }

  for (j = 0; j < i; j++) {
    str[j] = curr_str[i - j];
  }
  str[j] = '\0';
}
```

- Converts unsigned int to ASCII string
- Manual reverse construction without libraries

⚙️ **ADC1_Init()**

```
void ADC1_Init() {
  RCC_APB2ENR |= (1 << 9);      // Enable ADC1 clock
  ADC1_CR2 = 0; ADC1_CR1 = 0;
  ADC1_SMPR2 |= (7 << 0);       // Longest sample time (239.5 cycles)
  ADC1_CR2 |= (1 << 0);         // Turn on ADC1
  Delay_ms(1);
```

}

💉 Proteus Simulation Summary

- **ACS712** connected to battery through a load (resistor)
- Its output (VOUT) connected to **PA0**
- LCD connected to **GPIOA and GPIOB**
- **RV1** sets LCD contrast

✅ Output is real-time current (in Amps) displayed on LCD

✅ What You Learn from This Project

Topic	Description
Analog Measurement	Read current sensor using ADC
LCD Interface	Use GPIO + LCD commands to display strings
Voltage-to-Current	Convert voltage to Amps using ACS712 scale
Bypassing std libs	Manual string conversion without sprintf

Would you like:

- A more advanced version using floating-point display?
- To add energy (Watt * time) tracking?
- Or continue to **Project 21**?

Let me know!

Perfect! Let's now fully explain **Project 21: Garage Door Control System using IR Sensor and Motor** with STM32F103 and MikroC.

🚪 Project 21: Garage Door Automation

🎯 Objective:

- Automatically **open** the garage door when something is detected by the **IR sensor**
- Automatically **close** the door when nothing is detected
- Use **ADC** to read sensor signal and **GPIO** to control motor direction

---

🛠️ Hardware Components:

Component	Purpose
IR sensor	Detects obstacle (car/person)
STM32F103C6	Controls motor & reads sensor
Motor driver (L293D)	Drives motor forward/reverse
DC motor	Simulates garage door mechanism
LCD (optional)	Not used here but could show status

📜 Code Breakdown (MikroC)

🔸 Variable Declaration

unsigned int adc_value;

Stores the analog reading from the IR sensor (through PA0 ADC input).

🔸 GPIO & ADC Initialization

```
GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_0 | _GPIO_PINMASK_1);
// Set PB0 and PB1 as output pins for motor control

ADC_Set_Input_Channel(_ADC_CHANNEL_0);
// Configure PA0 as ADC input (IR sensor output)
```

🔁 Main Logic Loop

```
while(1) {
  adc_value = ADC1_Get_Sample(0);  // Read from channel 0 (PA0)
```

🔍 If Object is Detected

```
if(adc_value < 2000) {
  // Object is close (IR voltage low)
  GPIOB_ODR.B0 = 1;  // IN1 = HIGH (Motor Forward)
  GPIOB_ODR.B1 = 0;  // IN2 = LOW
}
```

How it works:

- IR sensor outputs **lower voltage** when object is detected
- You set motor to rotate **forward** to **open the door**

❌ If No Object Detected

```
else {
  GPIOB_ODR.B0 = 0;  // IN1 = LOW
  GPIOB_ODR.B1 = 1;  // IN2 = HIGH (Motor Reverse)
}
```

How it works:

- If no object is detected (voltage high), reverse motor to **close the door**

💉 ADC Value Mapping

- IR sensors like TCRT5000 or similar output **high voltage (>2.5V)** when nothing is detected, and **low voltage (<2.5V)** when object is near.
- 2000 corresponds to ~1.6V on a 12-bit ADC (0–4095 for 0–3.3V)

🖥️ Proteus Simulation Highlights

🧩 Components:

- **IR1** = IR obstacle sensor → output to PA0
- **U2** = L293D motor driver:
  - IN1 = PB0
  - IN2 = PB1
  - OUT1, OUT2 connected to DC motor
- **U1** = STM32F103C6

⚙️ Working Flow:

Sensor Status	ADC Value	PB0	PB1	Motor Direction	Door Action
Object Detected	< 2000	1	0	Forward	Open
No Object	≥ 2000	0	1	Reverse	Close

✅ What You Learn in This Project

Concept	Description
ADC + IR Sensor	Detect proximity with analog voltage
Motor Control	Drive direction using GPIO pins

Concept	Description
Basic Automation	Respond to sensor data to automate actions
GPIO Configuration	Controlling motor driver with STM32 pins

🚀 Enhancements You Can Add

- Add **LCD screen** to show “Opening...” or “Closing...”
- Add **limit switch** or **timer** to stop motor after door is fully open/closed
- Use **interrupts** or **PWM** to fine-tune motor control

Would you like to:

- Simulate this with an LCD status output?
- Move on to **Project 22**?
- See how to convert it into a mobile-controlled system?

Let me know!

Excellent! Let's now explain **Project 22: Stepper Motor Control Using STM32 and GPIO in MikroC**.

🌀 Project 22: Stepper Motor Control via GPIO

🎯 Objective:

Control the rotation of a **unipolar stepper motor** by energizing its coils in a specific sequence using **GPIO pins PA0 to PA3**.

🛠️ Hardware Components:

Component	Purpose
STM32F103C6	Controls GPIO outputs
ULN2003A Driver	Amplifies GPIO signals to drive motor
Stepper Motor (e.g. 28BYJ-48)	Motor to rotate in steps
External power	5V or 12V for motor
Crystal	8MHz external oscillator for STM32

📜 Full Code Explanation

🔸 main() function starts the execution:

```
void main() {
```

🔧 Pin Configuration

```
GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_0 | _GPIO_PINMASK_1 |
_GPIO_PINMASK_2 | _GPIO_PINMASK_3);
```

- Configure **PA0–PA3** as **digital output pins**
- These pins control the **4 inputs of ULN2003A** that drive the stepper motor

---

🔁 Stepper Motor Coil Energizing Sequence (4-Step Full Step Mode)

Each step energizes **2 coils at a time**, rotating the motor one step forward.

🔹 Step 1: Energize PA0 and PA3

```
GPIOA_ODR.B0 = 1;  // PA0 HIGH
GPIOA_ODR.B1 = 0;  // PA1 LOW
GPIOA_ODR.B2 = 0;  // PA2 LOW
GPIOA_ODR.B3 = 1;  // PA3 HIGH
Delay_ms(100);    // Wait 100ms
```

➞ This combination energizes the 1st and 4th coils.

🔹 Step 2: Energize only PA0

```
GPIOA_ODR.B0 = 1;  // PA0 HIGH
GPIOA_ODR.B1 = 0;  // PA1 LOW
GPIOA_ODR.B2 = 0;  // PA2 LOW
GPIOA_ODR.B3 = 0;  // PA3 LOW
Delay_ms(100);
```

➞ Only coil 1 is energized.

🔹 Step 3: Energize only PA1

```
GPIOA_ODR.B0 = 0;  // PA0 LOW
GPIOA_ODR.B1 = 1;  // PA1 HIGH
GPIOA_ODR.B2 = 0;  // PA2 LOW
GPIOA_ODR.B3 = 0;  // PA3 LOW
```

Delay_ms(100);

➞ Coil 2 is energized.

🔹 **Step 4: Energize PA1 and PA3**

```
GPIOA_ODR.B0 = 0;  // PA0 LOW
GPIOA_ODR.B1 = 1;  // PA1 HIGH
GPIOA_ODR.B2 = 0;  // PA2 LOW
GPIOA_ODR.B3 = 1;  // PA3 HIGH
Delay_ms(100);
```

➞ Coils 2 and 4 are energized.

🔁 **The while(1) loop keeps repeating this sequence**

Each pass of the loop rotates the motor **one full cycle through 4 steps**, generating continuous motion.

You can:

- Decrease Delay_ms() to make motor rotate faster
- Reverse the sequence to rotate the motor in **reverse direction**

💉 **Proteus Simulation Notes**

🧩 **Wiring Summary:**

- **PA0–PA3** → IN1–IN4 on ULN2003A
- **ULN2003A OUT1–OUT4** → Stepper Motor coil wires
- **+12V or +5V** to Vcc of motor
- Common GND between STM32 and driver

✅ **What You Learn from This Project**

Concept	Description
GPIO Output	Use of digital pins to control coils
Stepper Motor Phasing	Energize motor coils in sequence
ULN2003A Driver Usage	Transistor array to drive inductive loads
Timing Control	Using delays to control speed

🚀 **Suggested Enhancements**

- Add **push button** to start/stop rotation
- Add **potentiometer** to control delay (i.e., speed)
- Display direction/speed on **LCD**
- Use **timer interrupt** instead of Delay_ms()

Would you like to:

- Simulate reverse direction control?
- Extend this with a keypad-controlled rotation?
- Move to Project 23?

Let me know how you'd like to proceed!