Of course! For interfacing a quadrature encoder with your **STM32L432KC**, the most efficient and reliable method is to use a hardware timer in its dedicated **Encoder Mode**. This offloads all the decoding work from the CPU.

For easy setup, connect your encoder's **A** and **B** output channels to the following pins:

- **Channel A** to **PA0** (which can be `TIM2_CH1`)
- **Channel B** to **PA1** (which can be `TIM2_CH2`)

Using this hardware-based approach means you won't need complex interrupt handlers to decode the signals yourself. The timer will automatically increment or decrement its counter register based on the encoder's rotation.

The Recommended Method: Hardware Timer in Encoder Mode

This is the standard and most robust way to handle quadrature encoders on STM32 microcontrollers. It's resilient to high rotation speeds and uses virtually no CPU time.

How It Works

You configure a general-purpose timer (like `TIM2`) to act as an encoder interface. The timer's hardware monitors the `CH1` and `CH2` pins. Based on the sequence of rising and falling edges of the A and B signals, it automatically increments the timer's counter (`TIM2->CNT`) for clockwise rotation and decrements it for counter-clockwise rotation.

Your main application code simply needs to read this counter value whenever you want to know the encoder's position.

Reading the Encoder Position

After configuring `TIM2` in Encoder Mode (which is easily done using STM32CubeMX), your code to get the position is incredibly simple.

```
// This code assumes you have already configured TIM2 in Encoder Mode
// and started it with HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);

// Get the current encoder position
// The counter is a 16-bit or 32-bit register depending on the timer
int32_t get_encoder_position(void) {
    // Read the timer's counter register directly
    return (int16_t)TIM2->CNT; // Cast to signed 16-bit for TIM2
}

// You can also reset the position at any time
void reset_encoder_position(void) {
    TIM2->CNT = 0;
}
```

The Manual Method: Using External Interrupts

While not recommended for performance, I'm including this as you specifically asked for interrupt handlers. This approach uses External Interrupts (EXTI) on two GPIO pins and requires you to implement the decoding logic in software.

Pin Connections

You can use any two GPIO pins that can trigger interrupts. For this example, let's stick with **PA0** and **PA1**. You would configure them as inputs and enable interrupts on both rising and falling edges for both pins.

Interrupt Handler Code

The most robust way to manually decode the signals is with a state machine. The two signals (A and B) give you four possible states (00, 01, 10, 11). A valid rotation moves from one state to an adjacent one. By tracking the transition from the `old_state` to the `new_state`, you can determine the direction.

This code implements a lookup table for all 16 possible state transitions. It's fast and reliable.

```
#include "main.h" // Assuming HAL library usage

// Global variable to store the encoder's position.
// 'volatile' is crucial because it's modified within an ISR.
volatile int32_t encoder_position = 0;

// Function to be called from the interrupt handlers.
// It reads the pins and updates the position.
void update_encoder(void) {
    static uint8_t old_state = 0;

    // Read the current state of pins A and B.
    uint8_t pin_a_state = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0);
    uint8_t pin_b_state = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1);

    // Combine the two pin states into a single 2-bit value.
    uint8_t new_state = (pin_b_state << 1) | pin_a_state;

    // A lookup table to determine direction based on state transition.
    // Index is calculated as: (old_state << 2) | new_state
    // Values: 0 = no change, 1 = clockwise, -1 = counter-clockwise, 2 = error
    // Errors (e.g., jumping from state 00 to 11) are ignored.
    const int8_t transition_table[16] = {
      //              New State
      // Old State | 00  01  10  11
      /* 00 */       0, -1,  1,  0,
      /* 01 */       1,  0,  0, -1,
      /* 10 */      -1,  0,  0,  1,
      /* 11 */       0,  1, -1,  0
    };

    // Calculate the change in position
    int8_t change = transition_table[(old_state << 2) | new_state];
    encoder_position += change;

    // Save the current state for the next interrupt
    old_state = new_state;
}

/**
  * @brief  EXTI line detection callback.
  * @param  GPIO_Pin: Specifies the port pin connected to the EXTI line.
  * @retval None
  * * This single callback function is called by the HAL library for any EXTI event.
  * You must have configured PA0 and PA1 for EXTI in your setup.
  */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    // Check if the interrupt came from either of our encoder pins
    if (GPIO_Pin == GPIO_PIN_0 || GPIO_Pin == GPIO_PIN_1) {
        update_encoder();
    }
}
```

In summary, while the manual interrupt method works and is a great learning exercise, the **hardware timer method is superior** for its simplicity, performance, and reliability. I highly recommend using it for your project.