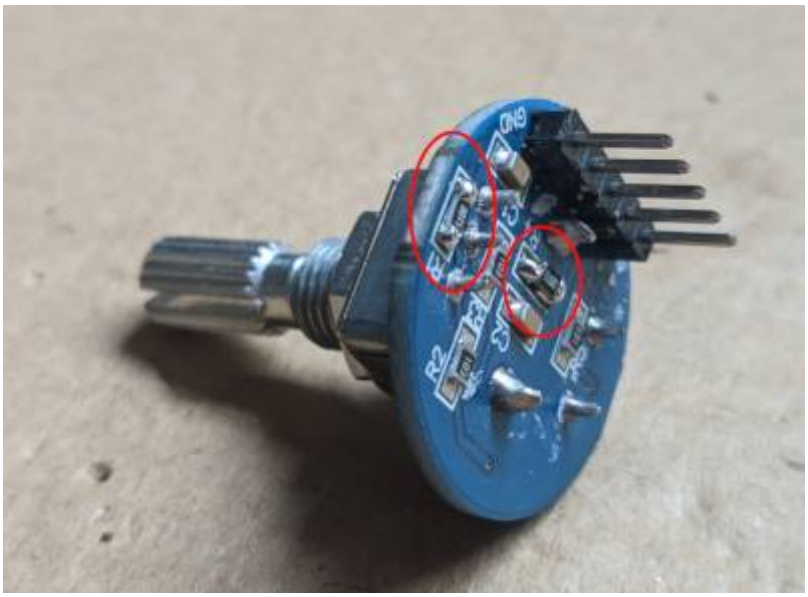
ESP32 - Rotary Encoder

How to reliably read rotary encoder on ESP32?

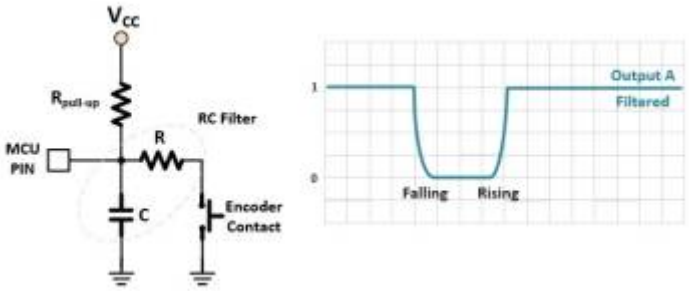
Background

For usage on my ESP32 projects, I ordered several rotary encoders from Ali Express. It appeared that all of these gave many erratic and unreliable readings. Indicating CW when turning CWW and visa versa. Approximately out out the 20-30 readings was erroneous. Initially I assumed contact bouncing, but the problem persisted after implementing an anti-bounce filter in the software. On a Raspberry this problem does not exist, so it seems that the ESP32 does not work well with this type of rotary encoders. This page describes both cause and solution for this problem.



Problem cause

It appears that the encoders I bought have RC filters on their output connections. Apparently, with the intention to eliminate contact bounce. However, side effect of this is that the rise and decay times of the rotary pulses do not have sharp edges, but curved edges. The image below is borrowed from www.allaboutcircuits.com (see links below) illustrates this. The ESP32 interrupt mechanism is not able to handle well interrupts and readings on slow edges. This resulted in erroneous readings. E.g. interrupts on negative edges showed a gpio value of 1 in the interrupt routine, where 0 is expected.



Hardware Modifications

The rotary encoders I bought appeared to have resistors of 10k in series with the rotary contact (value of R in picture above). After replacing these resistors to 501 Ohm, the readings improved significantly. The positions for S1 and S2 contacts on the back of the encoder are marked with red circles in the first image.

Reading Method 1

This is the most simple approach, requiring only 1 interrupt (either on S1 or S2). It appeared to work flawless. In this approach only of the S1 or S2 signals needs to be monitored. So, one interrupt on the negative edge is sufficient. When an interrupt occurs, compare the values of S1 and S2. If they are equal the rotation is on one direction. If they differ, the rotation is in the opposite direction.

```
typedef struct rotery_event {
  uint32_t gpio_value;
  int interupt;
} rotery_event;

#define MASK_PIN16  1<<16
#define MASK_PIN17  1<<17
#define MASK_PIN21  1<<21

// Static key definitions:
static int rotery_switch    = 0;
static int rotery_r1        = 0;
static int rotery_r2        = 0;
static int debounce_block   = 0;
int rotery_value            = 0;

int seq_idx                    = 0;
int seq_a                     = 0;
int seq_b                     = 0;

int rotery_value_right         = ROTERY_RIGHT;
int rotery_value_left         = ROTERY_LEFT;

static QueueHandle_t gpio_evt_queue = NULL;
```

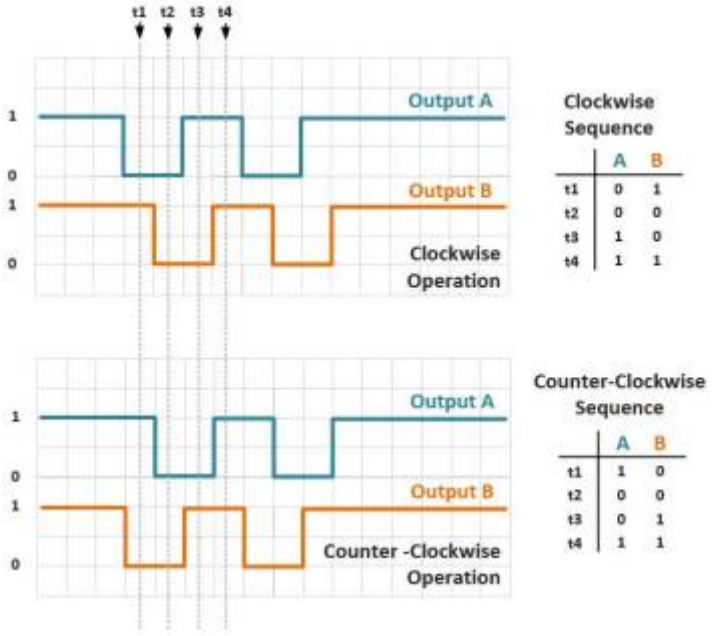
```
esp_timer_handle_t rotery_debounce_timer_handle;    // Timer to remove
(multiple fast GPIO interrupt updates)

portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
SemaphoreHandle_t xSemaphore;

//-------------------------------------------------------------
// gpio_isr_handler - Interrupt Handler
//-------------------------------------------------------------
static void IRAM_ATTR gpio_isr_handler( void* arg )
{
    uint32_t gpioValue;
    uint32_t returnValue;
    struct rotery_event re;

    xSemaphoreTakeFromISR( xSemaphore, &xHigherPriorityTaskWoken );

    if ( !debounce_block )
    {
        re.gpio_value = REG_READ(GPIO_IN_REG);
        re.interupt = *((int*)arg);

        if ( ((re.interupt == 16) && ((re.gpio_value & MASK_PIN16) == 0 ))
||
             ((re.interupt == 17) && ((re.gpio_value & MASK_PIN17) == 0 )) )

            debounce_block = 1;
            // Just post the interrupt event to the queue
            xQueueSendFromISR( gpio_evt_queue, &re, NULL );
        }
    }
    xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
}

//-------------------------------------------------------------
// gpio_event_task - Wait & procress GPIO Events from Queue
//-------------------------------------------------------------
static void gpio_event_task( void* arg )
{
    struct rotery_event re;
    for(;;)
    {
        if ( xQueueReceive( gpio_evt_queue, &re, portMAX_DELAY ))
        {
            int pin16 = (re.gpio_value & MASK_PIN16)>>16;
            int pin17 = (re.gpio_value & MASK_PIN17)>>17;
            int pin21 = (re.gpio_value & MASK_PIN21)>>21;

            if (pin16 == pin17)
                printf("intr [%d]  16[%d] 17[%d] 21[%d] --> LEFT\n",
re.interupt, pin16, pin17, pin21 );
```

```
            else
                printf("intr [%d]  16[%d] 17[%d] 21[%d] --> RIGHT\n",
re.interupt, pin16, pin17, pin21 );
            vTaskDelay(80 / portTICK_PERIOD_MS);

            xSemaphoreTake( xSemaphore, portMAX_DELAY );
            debounce_block = 0;
            xSemaphoreGive( xSemaphore );
        }
    }
}


// ---------------------------------------------------------------------------
-------------
// ROTERY_INIT
// ---------------------------------------------------------------------------
-------------
void rotery_init( int switch_gpio, int r1_gpio, int r2_gpio )
{
    gpio_config_t io_conf;

    xSemaphore = xSemaphoreCreateMutex();

        // Save the GPIO in the global variables
        rotery_switch     = switch_gpio;
        rotery_r1         = r1_gpio;
        rotery_r2         = r2_gpio;

        io_conf.mode = GPIO_MODE_INPUT;                    //set as input
mode
        io_conf.intr_type = GPIO_INTR_NEGEDGE;            //interrupt of
negative edge (pulldown by encoder)
        io_conf.pull_up_en = GPIO_PULLUP_DISABLE;        //disable pull-up
mode
        io_conf.pull_down_en = GPIO_PULLDOWN_DISABLE;    //disable pull-down
mode

        io_conf.pin_bit_mask = (1ULL<<rotery_switch);    //bit mask of the
pins that you want to set
        gpio_config( &io_conf );
        io_conf.pin_bit_mask = (1ULL<<rotery_r1);        //bit mask of the
pins that you want to set
        gpio_config( &io_conf );
        io_conf.pin_bit_mask = (1ULL<<rotery_r2);        //bit mask of the
pins that you want to set
        gpio_config( &io_conf );

        //create a queue to handle gpio event from isr
        gpio_evt_queue = xQueueCreate( 16, sizeof(struct rotery_event) );
```

```
        //start gpio task
        xTaskCreate( gpio_event_task, "gpio_task", 4096, NULL, 10, NULL );

        //hook isr handler for specific gpio pin
        gpio_install_isr_service( 0 );
        gpio_isr_handler_add( rotery_switch, gpio_isr_handler, (void*)
&rotery_switch );
        gpio_isr_handler_add( rotery_r1, gpio_isr_handler, (void*)
&rotery_r1 );
}
```

Reading Method 2

This is a more complete/correct approach, requiring 2 interrupts (both on S1 or S2). It should provide more reliability/resilience than method 1. The description below is borrowed from www.allaboutcircuits.com (see links below). A rotary encoder generates two output signals during rotation. Depending on the direction, one of the signals leads the other. You can see the output signal waveforms of an incremental rotary encoder and the expected bit sequence below.



Connecting S1 and S2 to 2 GPIO pins with interrupts triggered on both edges. With this we can track both S1 and S2 signals and detect a rotation when an expected sequence is received. As it can be seen from the waveform diagram, a clockwise motion generates A = ...0011... and B = ...1001... When we record both of the signals in bytes seqA and seqB by shifting in the last reading from right, we can compare these values and determine a new rotational step.

```
// Static key definitions:
static int rotery_switch    = 0;
static int rotery_r1        = 0;
static int rotery_r2        = 0;
static int debounce_block   = 0;
int rotery_value            = 0;
```

```
int seq_idx                    = 0;
int seq_a                     = 0;
int seq_b                     = 0;

int rotery_value_right         = ROTERY_RIGHT;
int rotery_value_left         = ROTERY_LEFT;

static QueueHandle_t gpio_evt_queue = NULL;

esp_timer_handle_t rotery_debounce_timer_handle;    // Timer to remove
(multiple fast GPIO interrupt updates)

portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
SemaphoreHandle_t xSemaphore;

//-------------------------------------------------------------
// gpio_isr_handler - Interrupt Handler
//-------------------------------------------------------------
static void IRAM_ATTR gpio_isr_handler( void* arg )
{
    uint32_t gpioValue;
    uint32_t returnValue;
    struct rotery_event re;

    xSemaphoreTakeFromISR( xSemaphore, &xHigherPriorityTaskWoken );

    if ( !debounce_block )
    {
        re.gpio_value = REG_READ(GPIO_IN_REG);
        re.interupt = *((int*)arg);

        if ( ((re.interupt == 16) ) || ((re.interupt == 17) ) )
        {
            debounce_block = 1;
            // Just post the interrupt event to the queue
            xQueueSendFromISR( gpio_evt_queue, &re, NULL );
        }
    }
    xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
}

//-------------------------------------------------------------
// gpio_event_task - Wait & procress GPIO Events from Queue
//-------------------------------------------------------------
static void gpio_event_task( void* arg )
{
    struct rotery_event re;
    for(;;)
    {
        if ( xQueueReceive( gpio_evt_queue, &re, portMAX_DELAY ))
```

```
        {
            int pin16 = (re.gpio_value & MASK_PIN16)>>16;
            int pin17 = (re.gpio_value & MASK_PIN17)>>17;
            int pin21 = (re.gpio_value & MASK_PIN21)>>21;

            if ( seq_idx != 0 )
            {
                seq_a=seq_a<<1;
                seq_b=seq_b<<1;
            }
            seq_a = seq_a | pin16;
            seq_b = seq_b | pin17;

            seq_idx++;

            if ( pin16 == 1 && pin17 == 1)
            {
                    if ( seq_a == 0x03 && seq_b == 0x09 )
                        printf("RIGHT\n" );
                    else if  ( seq_a == 0x09 && seq_b == 0x03 )
                        printf("LEFT\n" );
                    else
                        printf("ERROR seqA=[%x] seqB=%x]\n", seq_a, seq_b
);
                seq_idx = 0;
                seq_a = 0;
                seq_b = 0;
            }
            else if ( seq_idx == 4)
            {
                printf("SEQ ERROR seqA=[%x] seqB=%x]\n", seq_a, seq_b  );
                seq_idx = 0;
                seq_a = 0;
                seq_b = 0;
            }
            vTaskDelay(80 / portTICK_PERIOD_MS);
            xSemaphoreTake( xSemaphore, portMAX_DELAY );
            debounce_block = 0;
            xSemaphoreGive( xSemaphore );
        }
    }
}


// ----------------------------------------------------------------------------
-------------
// ROTERY_INIT
// ----------------------------------------------------------------------------
-------------
void rotery_init( int switch_gpio, int r1_gpio, int r2_gpio )
{
```

```
    gpio_config_t io_conf;

    xSemaphore = xSemaphoreCreateMutex();

    // Save the GPIO in the global variables
    rotery_switch     = switch_gpio;
    rotery_r1         = r1_gpio;
    rotery_r2         = r2_gpio;

    io_conf.mode = GPIO_MODE_INPUT;                    //set as input mode
    io_conf.intr_type = GPIO_INTR_ANYEDGE;            //interrupt of
negative edge (pulldown by encoder)
    io_conf.pull_up_en = GPIO_PULLUP_DISABLE;        //disable pull-up mode
    io_conf.pull_down_en = GPIO_PULLDOWN_DISABLE;    //disable pull-down
mode

    io_conf.pin_bit_mask = (1ULL<<rotery_switch);    //bit mask of the pins
that you want to set
    gpio_config( &io_conf );
    io_conf.pin_bit_mask = (1ULL<<rotery_r1);        //bit mask of the pins
that you want to set
    gpio_config( &io_conf );
    io_conf.pin_bit_mask = (1ULL<<rotery_r2);        //bit mask of the pins
that you want to set
    gpio_config( &io_conf );

    //create a queue to handle gpio event from isr
    gpio_evt_queue = xQueueCreate( 16, sizeof(struct rotery_event) );

    //start gpio task
    xTaskCreate( gpio_event_task, "gpio_task", 4096, NULL, 10, NULL );

    //hook isr handler for specific gpio pin
    gpio_install_isr_service( 0 );
    gpio_isr_handler_add( rotery_switch, gpio_isr_handler, (void*)
&rotery_switch );
    gpio_isr_handler_add( rotery_r1, gpio_isr_handler, (void*) &rotery_r1 );
    gpio_isr_handler_add( rotery_r2, gpio_isr_handler, (void*) &rotery_r2 );

    // Create timer for debouncing (multiple fast GPIO interrupt updates)
    const esp_timer_create_args_t rotery_debounce_timer_args = {
            .callback = &rotery_debounce_timer_callback,
            /* argument specified here will be passed to timer callback
function */
            .arg = (void*) rotery_debounce_timer_handle,
            .name = "rotery_debounce"
        };
    esp_timer_create( &rotery_debounce_timer_args,
&rotery_debounce_timer_handle );
}
```

Links

- www.allaboutcircuits.com