Introduction

The goal of this project is to make a simple blinky demonstration on a STM32F429 Discovery board which will be used for the final project. When the blue user push-button is pressed, it will generate an interrupt which starts a debouncing routine which will then toggle blinking between the red and green LEDs.

Setup

- Hardware : STM32F429I-DISC1
- Lenovo ThinkPad
- Ubuntu 20.04
- STM32CubeIDE Version 1.9.0
- https://github.com/kqr2/STemWin_HelloWorld

Software Overview

This demonstration is based on the STM32 demo application **STemWin_HelloWorld** which can be imported from STM32CubeIDE. The default application will display “Hello World” on the LCD screen and blink the red and green LEDs.

Other references used for this project:

- Example interrupt handler: https://deepbluembedded.com/stm32-external-interrupt-example-lab/
- Debouncing routine: https://docs.arduino.cc/built-in-examples/digital/Debounce

https://github.com/kqr2/STemWin_HelloWorld/blob/main/Drivers/BSP/STM32F429I-Discovery/stm32f429i_discovery.h

These are the port / pin assignments for the LEDs and user push-button. **EXTI0_IRQn** interrupt is used for the push-button.

```
// Green LED
#define LED3_PIN                                GPIO_PIN_13
#define LED3_GPIO_PORT                          GPIOG

// Red LED
#define LED4_PIN                                GPIO_PIN_14
#define LED4_GPIO_PORT                          GPIOG
```

```
// User Push-button
#define KEY_BUTTON_PIN                          GPIO_PIN_0
#define KEY_BUTTON_GPIO_PORT                    GPIOA
#define KEY_BUTTON_EXTI_IRQn                     EXTI0_IRQn
```

https://github.com/kqr2/STemWin_HelloWorld/blob/main/Core/Src/main.c

Several global variables were added to *main.c* to handle the interrupt, debounce the user push-button and switch toggling between the red and green LEDs.

```
// MES enhancements
uint8_t led_toggle = 1;

// Button interrupt has occurred
int button_pressed = 0;

// https://docs.arduino.cc/built-in-examples/digital/Debounce
int buttonState;             // the current reading from the input pin
int lastButtonState = -1;   // the previous reading from the input pin

// the following variables are unsigned longs because the time, measured in
// milliseconds, will quickly become a bigger number than can be stored in
an int.
int lastDebounceTime = 0;  // the last time the output pin was toggled
int debounceDelay = 2;  // the debounce time; increase if the output
flickers
```

https://github.com/kqr2/STemWin_HelloWorld/blob/main/Core/Src/stm32f4xx_it.c

In order to process the push-button interrupt, a handler must be added. In turn it calls the HAL interrupt handler for the push-button pin.

```
void EXTI0_IRQHandler(void)
{
      /* Button GPIO handler */
      HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}
```

https://github.com/kqr2/STemWin_HelloWorld/blob/main/Core/Src/main.c

When the user push-button is pressed, eventually the user callback is called which sets the **button_pressed** flag:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
   button_pressed = 1;
}
```

BSP_Config() initializes the LEDs and push-button.

```
/* Initialize STM32F429I-DISCO's LEDs */
  BSP_LED_Init(LED3);
  BSP_LED_Init(LED4);

  /* Initialize user button to cause an interrupt */
  BSP_PB_Init(BUTTON_KEY, BUTTON_MODE_EXTI);
```

The TIM3 timer is initialized such that the period is about 0.1 seconds. Prescaler sets the clock to about 10 KHz and the period is set to about 1000 / 10000 or 0.1 seconds.

```
 /* Compute the prescaler value to have TIM3 counter clock equal to 10 KHz
*/
  uwPrescalerValue = (uint32_t) ((SystemCoreClock /2) / 10000) - 1;

  /* Set TIMx instance */
  TimHandle.Instance = TIM3;

  /* Initialize TIM3 peripheral as follows:
      + Period = 1000 - 1
      + Prescaler = ((SystemCoreClock/2)/10000) - 1
      + ClockDivision = 0
      + Counter direction = Up
  */
  TimHandle.Init.Period = 1000 - 1;
```

Every 0.1 seconds the user timer callback is executed:

```
/**
  * @brief  Period elapsed callback in non blocking mode
  * @param  htim: TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  BSP_Background();
}
```

Which in turn calls the BSP_Background() code which performs the main user application:

- Toggle green or red LEDs
- Check if button_pressed flag is set
- If so, debounce the push-button
- If the push-button is successfully debounced then switch between the green or red LED to toggle

```
void BSP_Background(void)
{
      if (led_toggle) {
            // Green LED
            BSP_LED_Toggle(LED3);
            BSP_LED_Off(LED4);
      }
      else {
            // Red LED
            BSP_LED_Toggle(LED4);
            BSP_LED_Off(LED3);
      }

      if (button_pressed) {
        // read the state of the switch into a local variable:
        int button = BSP_PB_GetState(BUTTON_KEY);

        // check to see if you just pressed the button
        // (i.e. the input went from LOW to HIGH), and you've waited long
enough
        // since the last press to ignore any noise:

        // If the switch changed, due to noise or pressing:
        if (button != lastButtonState) {
```

```
          // reset the debouncing timer
          lastDebounceTime = 0;
        }

        if (++lastDebounceTime > debounceDelay) {
          // whatever the reading is at, it's been there for longer than
the debounce
          // delay, so take it as the actual current state:

              buttonState = button;
            if (buttonState) {
            // Toggle LED blinking state
              led_toggle = 1-led_toggle;
            }

          lastDebounceTime = 0;

          // reset button interrupt flag
          button_pressed = 0;
        }

        lastButtonState = button;
      }

  /* Capture input event and update cursor */
  if(GUI_Initialized == 1)
  {
    BSP_Pointer_Update();
  }
}
```

Further Investigations

Investigate further, using the processor manual:

- What are the hardware registers that cause the LED to turn on and off? (From the processor manual, don't worry about initialization.)
- What are the registers that you read in order to find out the state of the button?
- Can you read the register directly and see the button change in a debugger or by printing out the value of the memory at the register's address?

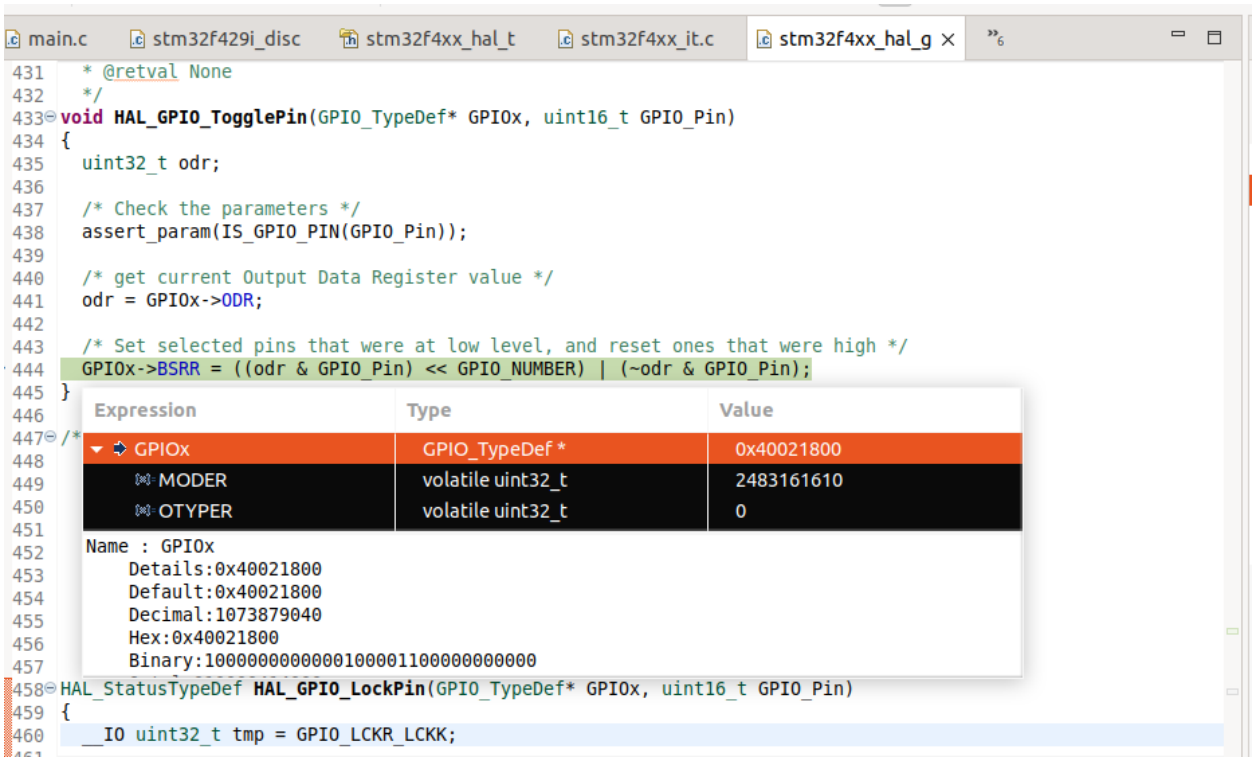
[STM32F429 Reference Manual]

```
// Green LED
#define LED3_PIN                                GPIO_PIN_13
#define LED3_GPIO_PORT                          GPIOG

// Red LED
#define LED4_PIN                                GPIO_PIN_14
#define LED4_GPIO_PORT                          GPIOG

// User Push-button
#define KEY_BUTTON_PIN                          GPIO_PIN_0
#define KEY_BUTTON_GPIO_PORT                    GPIOA
#define KEY_BUTTON_EXTI_IRQn                    EXTI0_IRQn
```

GPIO Port Memory Map:

0x4002 2800 - 0x4002 2BFF	GPIOK	AHB1	*Section 8.4.11: GPIO register map on page 287*
0x4002 2400 - 0x4002 27FF	GPIOJ		
0x4002 2000 - 0x4002 23FF	GPIOI		
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		*Section 8.4.11: GPIO register map on page 287*
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		

For toggling LEDs on Port G we can see the address 0x40021800 in the debugger.



8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..I/J/K)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)

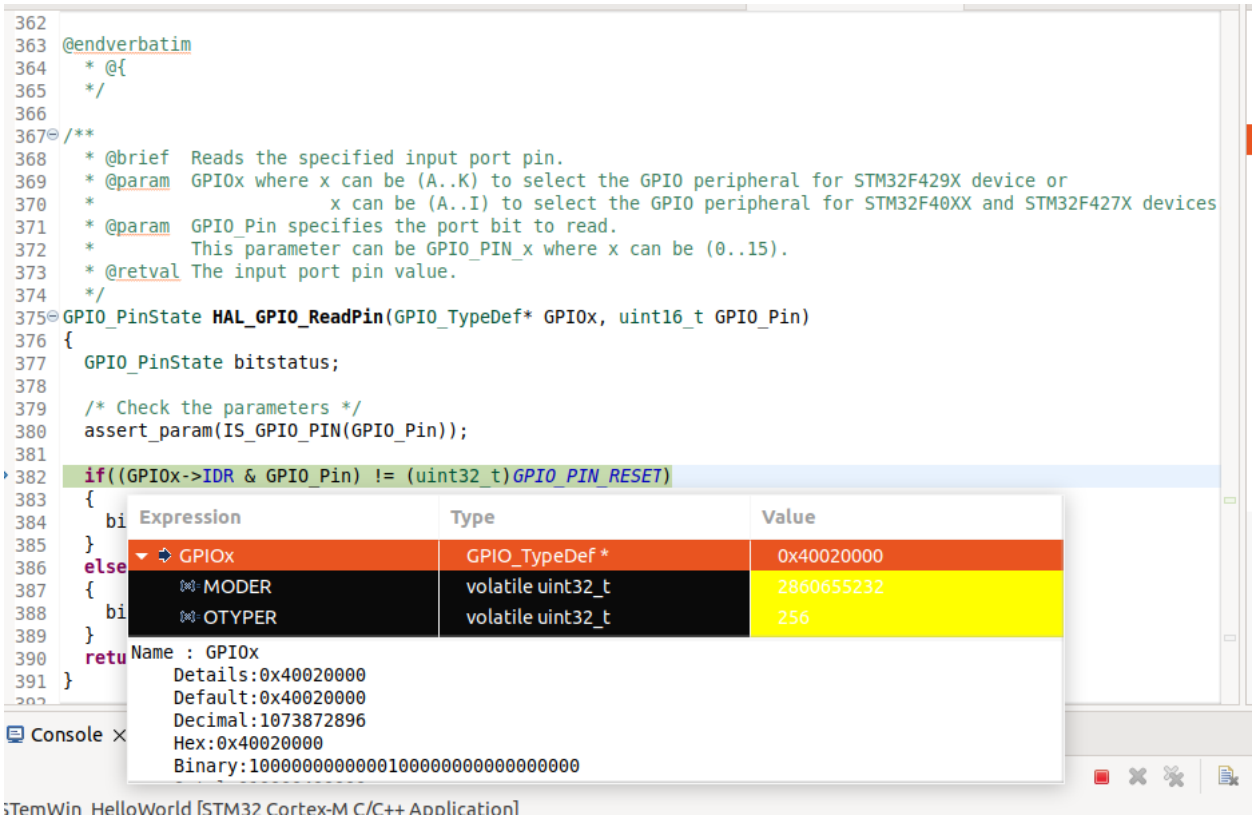
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Sets the corresponding ODRx bit

The BSRR is the Bit Set / Reset Register which actually toggles the pin. The effective address for GPIOG BSRR register is **0x40021818.**

For reading the user push-button on Port A we can see the address 0x4002000 in the debugger.



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

Address offset: 0x10

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

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

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

Bits 15:0 **IDRy**: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

The IDR is the Input Data Register which actually reads the value of the pin. The effective address for GPIOA IDR register is **0x40020010.**