西安電子科技大學
XIDIAN UNIVERSITY

Xidian Universtiy

2022 / EE205062

Solutions of **Travaux Pratiques**

Course: Microprocessor II

Author
KANG Gaalok

Professor
Yoann CHARLON
Université Côte D'Azur
YANG Xi
Xidian University

Rev. Date
June 14, 2022

March 24 – June 15

Contents

1 GPIO / LEDs

In this section, we will use mainly 2 methods to light the LEDs through the GPIO port:

- by operating the registers directly;
- by using the *standard drivers library* CMSIS.

By referring to the technical manual[1], we know that there are 4 LEDs we can program,

- LD3: orange LED, connected to the I/O PD13;
- LD4: green LED, connected to the I/O PD12;
- LD5: red LED, connected to the I/O PD14;
- LD6: blue LED, connected to the I/O PD15.

We will take LD4 as an example.

For the GPIO, it have 3 output type: pull-up, pull-down and float.

The pull-up/down type to ensure that if there is nothing connected to the pin and your program reads the state of the pin, will it be high (pulled to VCC, pull-up) or low (pulled to ground, pull-down).

In the internal circuit, these types are controled by a resistor. A low resistor value is called a strong pull-up (more current flows), a high resistor value is called a weak pull-up (less current flows).

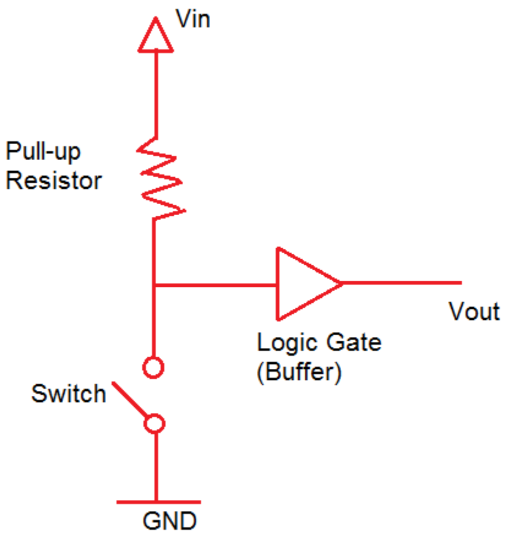

Figure 1: Pull-up Resistor Schematic

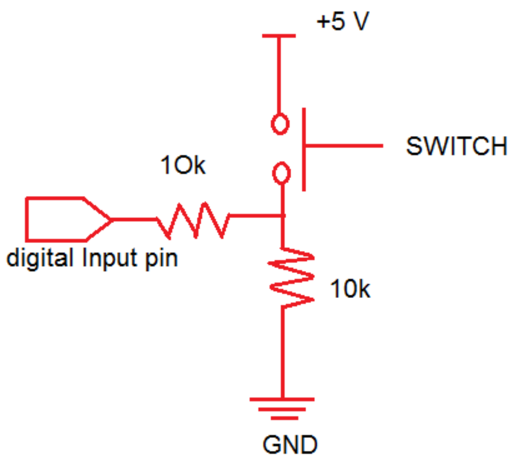

Figure 2: Pull-down Resistor Schematic

[1] UM1472, page 18

1.1 Operate the Registers

We mainly refer the Reference Manual[2].

To **initialize** the LD4, we need to configure the registers :

- `RCC_AHB1ENR` bit 3 SET 1, to active clock of port D;
- `GPIOD_MODER` bit [25,24] SET [01], to configure pin 12 *Mode* as **General purpose output mode**;
- `GPIOD_OTYPER` bit 12 SET 0, to configure pin 12 *OutputType* as **Output push-pull**;
- `GPIOD_OSPEEDR` bit [25,24] SET [10], to configure pin 12 *OutputSpeed* as **High speed**;
- `GPIOD_PUPDR` bit [25,24] SET [10], to configure pin 12 *Pull-up/down* as **Pull-down**.

To SET or CLEAR 1 bit on bit x, we use

```
    Register |=  (1 << x);   //SET
    Register &= ~(1 << x);  //CLEAR
```

Attention If we SET multi-bits on bit [y...x], we must CLEAR bit [y...x] first!

To **light** the LED LD4 on or off, we configure the register

- `GPIOD_BSRR` bit 12 (BS12) SET 1, to SET pin 12 and light LD4 on;
- `GPIOD_BSRR` bit 28 (BR12) SET 1, to CLEAR pin 12 and light LD4 off.

Attention If both BSx and BRx are set, BSx has priority!

1.2 Use the Functions in CMSIS

Attention To ensure that we can use the drivers library, we need to add some files to our project.

We use these functions:

- `LED_Initialize();`
- `LED_On();`
- `LED_Off()`.

```
/*-----------------------------------
 * LED_Initialize:  Initialize LEDs
 * Parameters: (none)
 * Return:     (none)
 *-----------------------------------*/
void LED_Initialize (void);

/*-----------------------------------
```

[2] RM0090

```
 * LED_On: Turns on requested LED
 * Parameters:  num - LED number
 * Return:      (none)
 *----------------------------------*/
void LED_On (uint32_t num);

/*----------------------------------
 * LED_Off: Turns off requested LED
 * Parameters:  num - LED number
 * Return:      (none)
 *----------------------------------*/
void LED_Off (uint32_t num);
```

2 Timer

In this section, we blink the LEDs with the frequency = 1Hz (0.5s for light on and 0.5s for light off). We need to find a clock frequency = 2Hz and change the LED state in a periode,

We use `TIM3` which the clock frequency = 84MHz. To set the new frequency = 2Hz, we need to divide it by 42M. It means that the interruption of `TIM3` will be active automatically every 0.5 second.

Set prescalar `PSC` = (10k-1) and counter periode `ARR` = (4200-1). Then we can get this new frequency.

$$f_{new} = \frac{f_{CLK}}{(PSC+1)(ARR+1)}$$

2.1 Operate the Registers

To **initialize** the `TIM3`, we need to configure the registers :

- `RCC_APB1ENR` bit 1 SET 1, to active the clock of `TIM3`;
- `TIM3_CR1` SET 0;
- `TIM3_PSC` SET (10 000 - 1) and `TIM3_ARR` SET (4200 - 1), to configure the scalar of the frequency;
- `TIM3_DIER` bit 0 SET 1, to configure the update interruption mode;
- `TIM3_CR1` bit 0 SET 1, to enable all the settings.

To **initialize** the Interruption, we need to configure the registers :

- `NVIC_ISER[0]` bit 29 SET 1.

Then, in the IRQ of the NVIC, first, we need to check whether it is in the interruption. Use `TIM3_SR` bit 0 as a flag, if it is **1**, then in the interruption. After we change the state of the LED, we CLEAR the flag by SETTING `TIM3_SR` bit 0 to 0.

Attention The function name of TIM3's IRQ is `TIM3_IRQHandler`!

Attention In the IRQ, we can't CLEAR the flag of interruption first!

2.2 Use the Functions in CMSIS

To **initialize** the `TIM3`, we use these functions:

```
/**
* @brief  Enables or disables the Low Speed APB (APB1) peripheral clock.
* @param  RCC_APB1Periph: specifies the APB1 peripheral to gates its
    clock.
*           This parameter can be any combination of the following values:
*             @arg RCC_APB1Periph_TIM3:   TIM3 clock
*            ...  ...  ...  ...  ...  ...  ...  ...  ...
```

```
* @param  NewState: new state of the specified peripheral clock.
*          This parameter can be: ENABLE or DISABLE.
* @retval None
*/
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState
    NewState);

typedef struct
{
  uint16_t TIM_Prescaler;
  uint16_t TIM_CounterMode;
  uint32_t TIM_Period;
  uint16_t TIM_ClockDivision;
  uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;

/**
* @brief  Initializes the TIMx Time Base Unit peripheral according to
*         the specified parameters in the TIM_TimeBaseInitStruct.
* @param  TIMx: where x can be  1 to 14 to select the TIM peripheral.
* @param  TIM_TimeBaseInitStruct:
*         pointer to a TIM_TimeBaseInitTypeDef structure that contains
*         the configuration information for the specified TIM peripheral.
* @retval None
*/
void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef*
    TIM_TimeBaseInitStruct);

/**
* @brief  Enables or disables the specified TIM interrupts.
* @param  TIMx: where x can be 1 to 14 to select the TIMx peripheral.
* @param  TIM_IT: specifies the TIM interrupts sources to be enabled or
    disabled.
*          This parameter can be any combination of the following values:
*            @arg TIM_IT_Update: TIM update Interrupt source
*            ...  ...  ...  ...  ...  ...  ...  ...  ...
* @param  NewState: new state of the TIM interrupts.
*          This parameter can be: ENABLE or DISABLE.
* @retval None
*/
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState
    NewState);

/**
* @brief  Enables or disables the specified TIM peripheral.
* @param  TIMx: where x can be 1 to 14 to select the TIMx peripheral.
* @param  NewState: new state of the TIMx peripheral.
*          This parameter can be: ENABLE or DISABLE.
```

```
* @retval None
*/
void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);
```

To **initialize** the NVIC, we use this function:

```
void NVIC_EnableIRQ (IRQn_Type IRQn);
```

To check the interruption states, we use these functions:

```
/**
* @brief  Checks whether the specified TIM flag is set or not.
* @param  TIMx: where x can be 1 to 14 to select the TIM peripheral.
* @param  TIM_FLAG: specifies the flag to check.
*          This parameter can be one of the following values:
*            @arg TIM_FLAG_Update: TIM update Flag
*            @arg TIM_FLAG_CC1: TIM Capture Compare 1 Flag
*            @arg TIM_FLAG_CC2: TIM Capture Compare 2 Flag
*            @arg TIM_FLAG_CC3: TIM Capture Compare 3 Flag
*            @arg TIM_FLAG_CC4: TIM Capture Compare 4 Flag
*            @arg TIM_FLAG_COM: TIM Commutation Flag
*            @arg TIM_FLAG_Trigger: TIM Trigger Flag
*            @arg TIM_FLAG_Break: TIM Break Flag
*            @arg TIM_FLAG_CC1OF: TIM Capture Compare 1 over capture Flag
*            @arg TIM_FLAG_CC2OF: TIM Capture Compare 2 over capture Flag
*            @arg TIM_FLAG_CC3OF: TIM Capture Compare 3 over capture Flag
*            @arg TIM_FLAG_CC4OF: TIM Capture Compare 4 over capture Flag
* @retval The new state of TIM_FLAG (SET or RESET).
*/
FlagStatus TIM_GetFlagStatus(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);

/**
* @brief  Clears the TIMx's pending flags.
* @param  TIMx: where x can be 1 to 14 to select the TIM peripheral.
* @param  TIM_FLAG: specifies the flag bit to clear.
* @retval None
*/
void TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);
```

3 PWM

In this section, we will know how to calculate the parameter of the PWM.

PWM is one of the application of `TIM`. Thus, to initialize the PWM, we must initialize `TIM` first. The functions we have already know. We take `TIM4` as an example.

```
void TIM4_Initialize(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructrue;
        // ENABLE RCC_APB1 for TIM4
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);
        // PCLK/(PSC+1) = 84MHz/1 = 84MHz
    TIM_TimeBaseInitStructrue.TIM_Prescaler=0;
        // 84MHz/(ARR+1) = 84MHz/8400 = 10kHz
    TIM_TimeBaseInitStructrue.TIM_Period=8399;
    TIM_TimeBaseInitStructrue.TIM_CounterMode=TIM_CounterMode_Up;
        // default: DIV2
    TIM_TimeBaseInitStructrue.TIM_ClockDivision=TIM_CKD_DIV1;
        // SET TIM4 PARAM
    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseInitStructrue);
}
```

For the CounterMode, there are mainly 2 kinds of mode: Counter-aligned mode and Edge-aligned mode. More detailed can be find in Timing control and PWM.

Attention The CenterAlignedMode1 will use the driver frequency twice than CenterAlignedMode3 if they realize the same frequency. And it will delay a quarter of a periode.

To **initialize** the PWM, to calculate the parameter of Pluse, we use

$$Pulse = \frac{(ARR + 1) \times DutyCycle}{100} - 1$$

and then we will use these functions:

```
typedef struct
{
    uint16_t TIM_OCMode;
    uint16_t TIM_OutputState;    // Normally we SET TIM_OutputState_Enable
    uint16_t TIM_OutputNState;
    uint32_t TIM_Pulse;
    uint16_t TIM_OCPolarity;     // Normally we SET TIM_OCPolarity_High
    uint16_t TIM_OCNPolarity;
    uint16_t TIM_OCIdleState;
    uint16_t TIM_OCNIdleState;
} TIM_OCInitTypeDef;

/**
* @brief  Initializes the TIMx Channel1 according to the specified
   parameters in
*         the TIM_OCInitStruct.
```

```
* @param  TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM
   peripheral.
* @param  TIM_OCInitStruct: pointer to a TIM_OCInitTypeDef structure that
    contains
*          the configuration information for the specified TIM peripheral.
* @retval None
*/
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);

/**
* @brief  Enables or disables the TIMx peripheral Preload register on
   CCR1.
* @param  TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM
   peripheral.
* @param  TIM_OCPreload: new state of the TIMx peripheral Preload
   register
*           This parameter can be one of the following values:
*             @arg TIM_OCPreload_Enable
*             @arg TIM_OCPreload_Disable
* @retval None
*/
void TIM_OC1PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
```

Finally, we configure the GPIO.

```
void GPIO_AF_Initialize(void){
        // ENABLE RCC AHB1
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD,ENABLE);
        // ENABLE mode alternate function / TIM4
        GPIO_PinAF(GPIOD,12,GPIO_AF_TIM4);


        // CONFIG mode: Alternate Function, Output Push Pull, Output
           Speed 100MHz, et no Pull up / down
        GPIO_PinConfigure(GPIOD,12, GPIO_MODE_AF,
                                GPIO_OUTPUT_PUSH_PULL,
                                GPIO_OUTPUT_SPEED_100MHz,
                                GPIO_NO_PULL_UP_DOWN);
}
```

4 UART

In this section, we will try to know a common communication protocol: **Serial Port**, also named **USART** (Universal Synchronous/Asynchronous Receiver-Transmitter). We take **UART** as an example.

4.1 Interface and Parameters of UART

The two signals of each UART[3] device are named:

- Transmitter (Tx);
- Receiver (Rx).

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication. The transmitting UART is connected to a controlling data bus that sends data in a parallel form. From this, the data will now be transmitted on the transmission line (wire) serially, bit by bit, to the receiving UART. This, in turn, will convert the serial data into parallel for the receiving device.

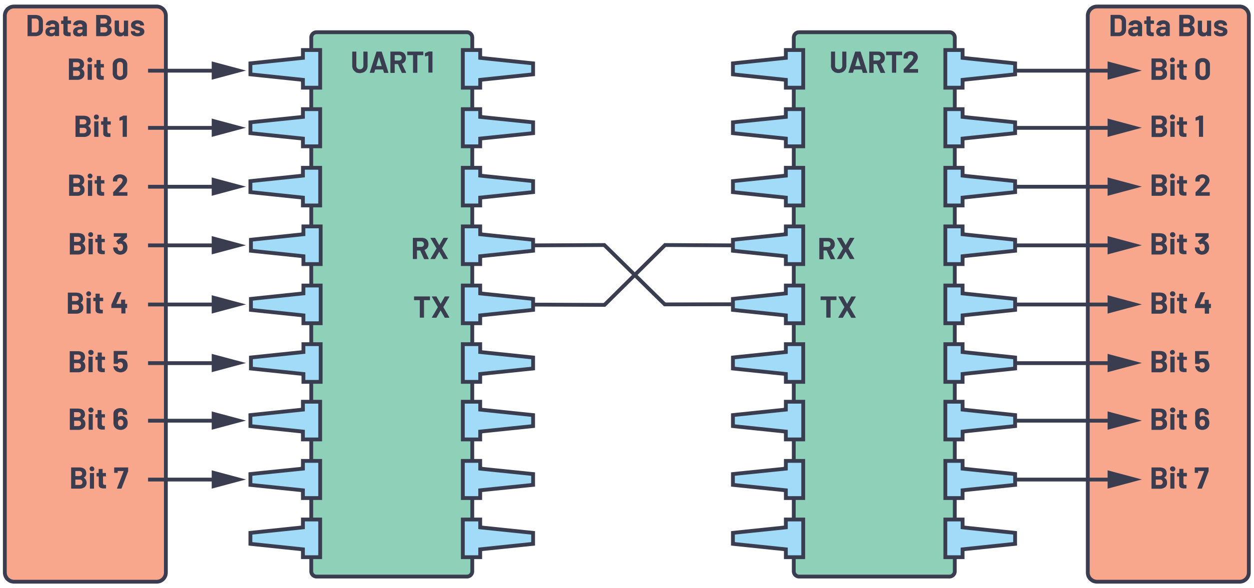

Figure 3: UART with data bus.

And there are several parameters of a UART in Table 1.

4.2 Use the Functions in CMSIS

First we need to **Initialize** the `USART3`. We will use these functions.

[3] More Detailed Knowledge

Parameters	Options
BaudRate	9600,115200,...
StartBit	1
WordLength	5 to 9
ParityBit	0, 1
StopBit	1, 2
...	...

Table 1: Parameters of A UART

```c
/**
* @brief  Enables or disables the Low Speed APB (APB1) peripheral clock.
* @param  RCC_APB1Periph: specifies the APB1 peripheral to gates its
  clock.
*           This parameter can be any combination of the following values:
*             @arg RCC_APB1Periph_USART3: USART3 clock
*             ...   ...   ...   ...   ...   ...   ...
* @param  NewState: new state of the specified peripheral clock.
*           This parameter can be: ENABLE or DISABLE.
* @retval None
*/
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState
  NewState);

typedef struct
{
  uint32_t USART_BaudRate;
  uint16_t USART_WordLength;
  uint16_t USART_StopBits;
  uint16_t USART_Parity;
  uint16_t USART_Mode;
  uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;

/**
* @brief  Initializes the USARTx peripheral according to the specified
*         parameters in the USART_InitStruct .
* @param  USARTx: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the
*         USART or UART peripheral.
* @param  USART_InitStruct: pointer to a USART_InitTypeDef structure that
*         contains the configuration information for the specified USART
*         peripheral.
* @retval None
*/
void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef*
  USART_InitStruct);
```

```c

/**
* @brief  Enables or disables the specified USART peripheral.
* @param  USARTx: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the
*         USART or UART peripheral.
* @param  NewState: new state of the USARTx peripheral.
*          This parameter can be: ENABLE or DISABLE.
* @retval None
*/
void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
```

Because the ports of `USART3 (Tx,Rx)` are connected to the `GPIO`, we need to **Initialize** the `GPIO`, and set the mode of GPIO to the `Alternate Function of USART3`. These steps we have already known.

Attention For the STM32F407 DISC Board, the pins of USART3 are setted: Tx->PB10, Rx->PB11.

As for the transmit and receive data:

```c
/**
* @brief  Transmits single data through the USARTx peripheral.
* @param  USARTx: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the
*         USART or UART peripheral.
* @param  Data: the data to transmit.
* @retval None
*/
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);

/**
* @brief  Returns the most recent received data by the USARTx peripheral.
* @param  USARTx: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the
*         USART or UART peripheral.
* @retval The received data.
*/
uint16_t USART_ReceiveData(USART_TypeDef* USARTx)
```

Attention These functions are used to transmit or receive A data.

Attention When we transmit and receive data, we must check the status and do the "hold on" function.

```c
/**
* @brief  Checks whether the specified USART flag is set or not.
* @param  USARTx: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the
*         USART or UART peripheral.
* @param  USART_FLAG: specifies the flag to check.
*          This parameter can be one of the following values:
*            @arg USART_FLAG_TXE:  Transmit data register empty flag
*            @arg USART_FLAG_TC:   Transmission Complete flag
```

```
*              @arg USART_FLAG_RXNE: Receive data register not empty flag
* @retval The new state of USART_FLAG (SET or RESET).
*/
FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint16_t USART_FLAG
    );
```

4.3 Verify on Board

To test the program, you need to use a CONVERTER. And maybe you have to INSTALL a DRIVER of CH341/CH340.

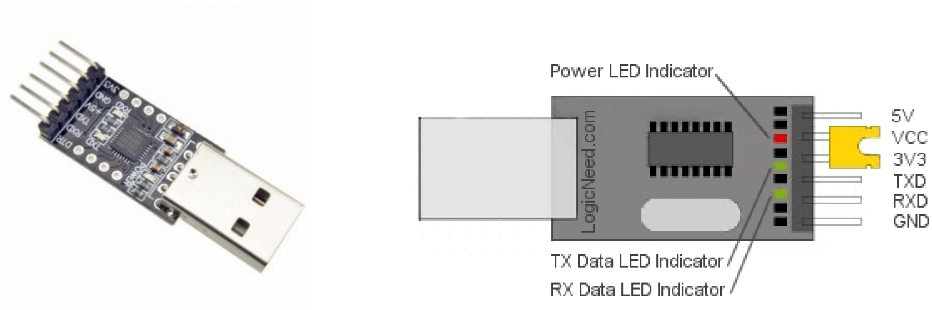

Figure 4: Converter

Attention The connection between the board and the converter is shown as Table 2.

Board PIN	Converter PIN
Tx(PB10)	Rx
Rx(PB11)	Tx
GND	GND

Table 2: Connection

5 NVIC

Attention You can not use two pins on one line simultaneously:

- PA0 and PB0 and PC0 and so on, are connected to `Line0`, so you can use only one pin at one time to handle interrupt;
- PA0 and PA5 are connected to different lines, they can be used at the same time.

STM32 Interrupts Tutorial