Embedded System

Prof. Mehdi Pirahandeh

Inha University

Email: *mehdi@inha.ac.kr*

Cortex
Intelligent Processors by ARM

인하대학교

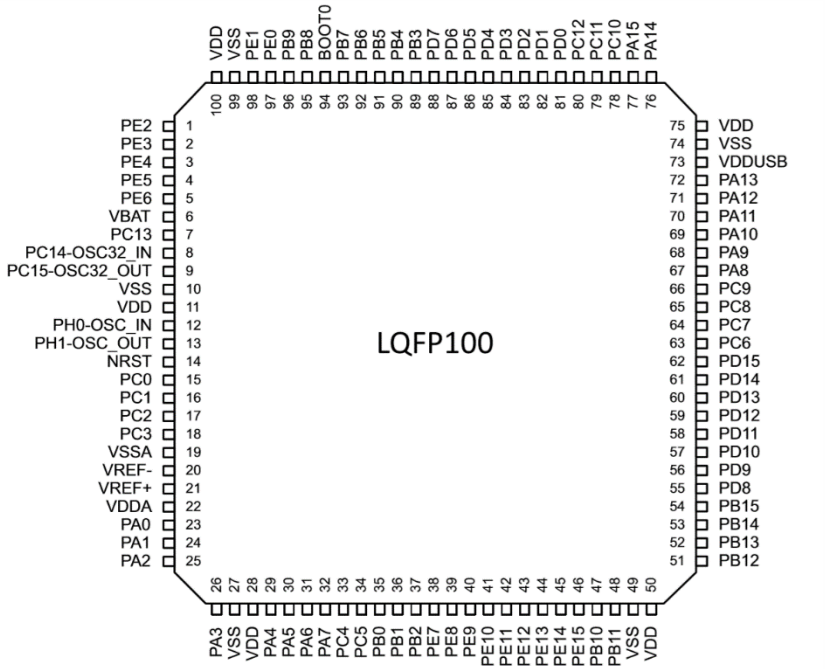
Lab 2. GPIO and USART

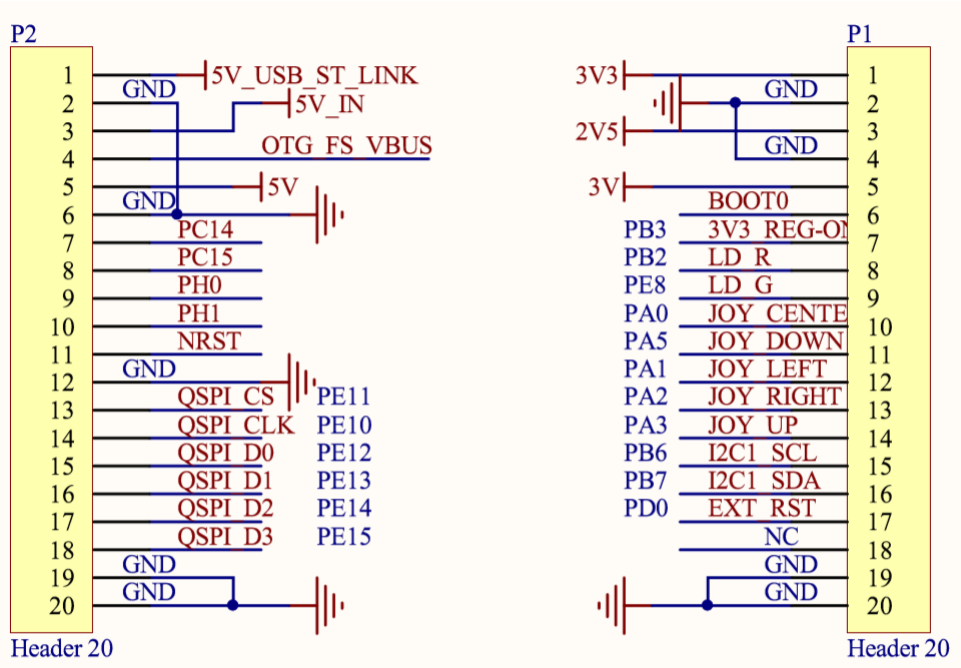
- GPIO
- USART
- Group Tasks

GPIO

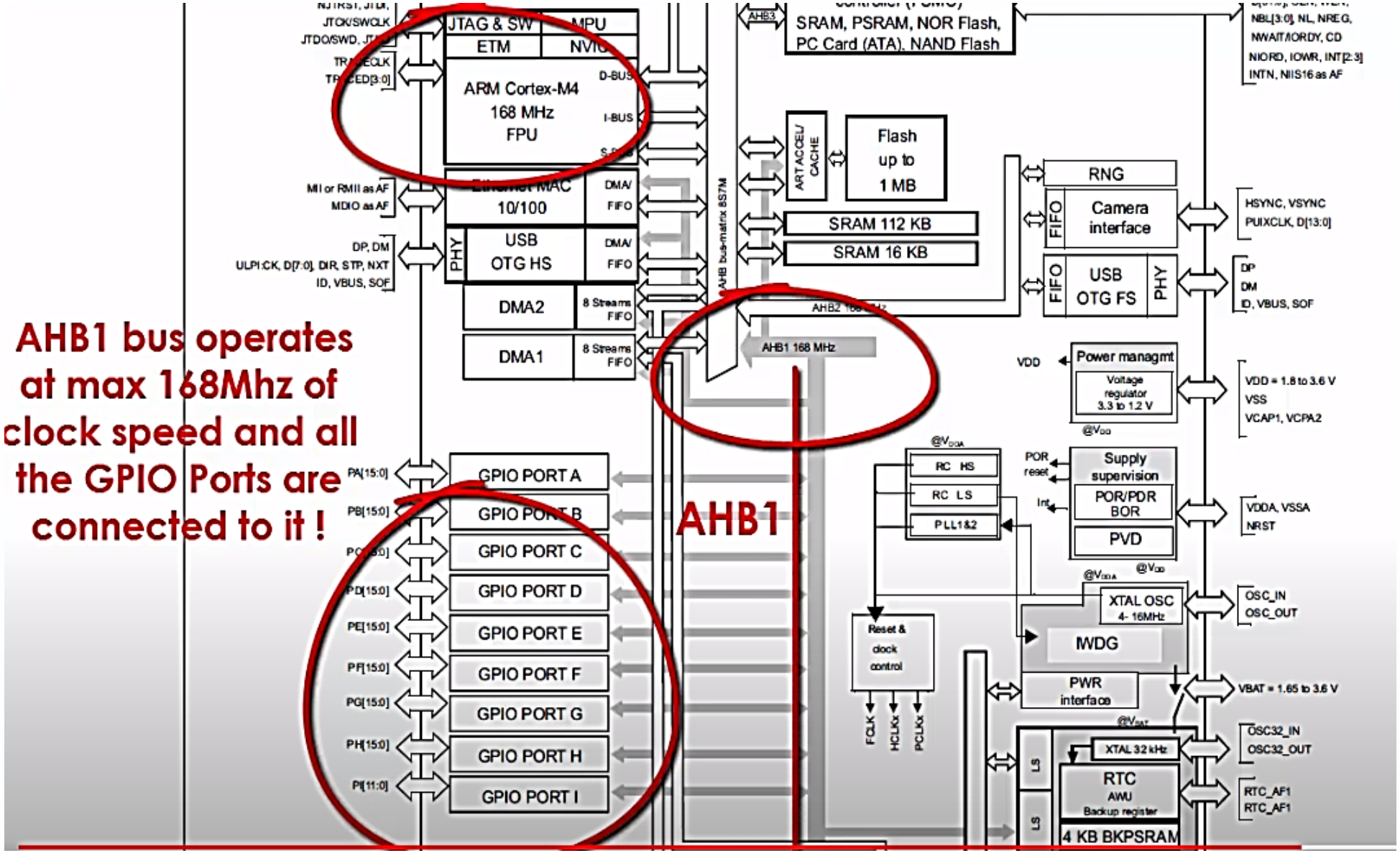
GPIO

- GPIO stands for General Purpose Input/Output.
- Serve as general digital input or output ports(as its name implies).
- Also can be configured to do analog or alternate functions.
- 5 Ports(A ~ E), each port has 16 pins(0 ~ 15) in STM32L476VG.
- 17 (+ 28 when you pull out the on-board LCD) pins on 32L476G Discovery board.



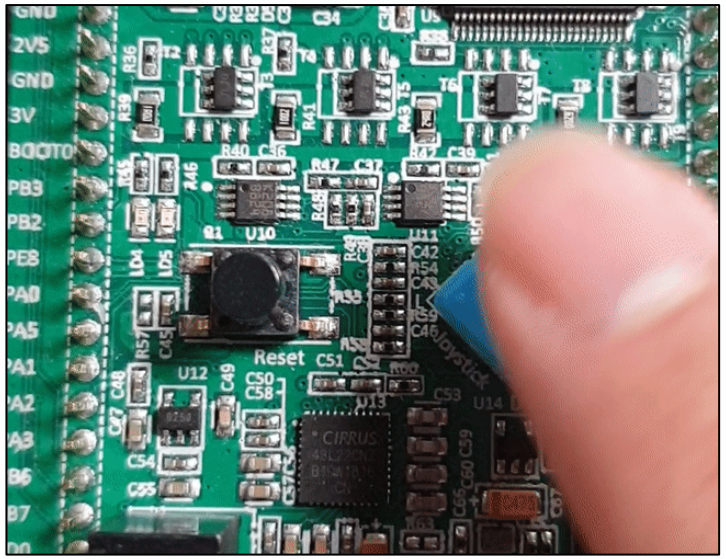


GPIO


AHB1 bus operates at max 168Mhz of clock speed and all the GPIO Ports are connected to it !
AHB1
JTAG & SW
MPU
ETM
ARM Cortex-M4
168 MHz
FPU
D-BUS
I-BUS
Ethernet MAC
10/100
DMA/
FIFO
MII or RMII as AF
MDIO as AF
USB
OTG HS
PHY
DP, DM
ULPI:CK, D[7:0], DIR, STP, NXT
ID, VBUS, SOF
DMA2
DMA1
8 Streams
FIFO
AHB bus-matrix 8S7M
AHB3
SRAM, PSRAM, NOR Flash,
PC Card (ATA), NAND Flash
NBL[3:0], NL, NREG,
NWAIT/IORDY, CD
NIORD, IOWR, INT[2:3]
INTN, NIIS16 as AF
ART ACCEL/
CACHE
Flash
up to
1 MB
SRAM 112 KB
SRAM 16 KB
AHB1 168 MHz
RNG
FIFO
Camera
interface
HSYNC, VSYNC
PUIXCLK, D[13:0]
USB
OTG FS
DP
DM
ID, VBUS, SOF
Power managmt
Voltage
regulator
3.3 to 1.2 V
VDD
VDD = 1.8 to 3.6 V
VSS
VCAP1, VCPA2
RC HS
RC LS
PLL1&2
Supply
supervision
POR/PDR
BOR
PVD
POR
reset
Int
VDDA, VSSA
NRST
XTAL OSC
4- 16MHz
OSC_IN
OSC_OUT
IWDG
Reset &
clock
control
PWR
interface
VBAT = 1.65 to 3.6 V
FCLK
HCLKx
PCLKx
XTAL 32 kHz
OSC32_IN
OSC32_OUT
RTC
AWU
Backup register
RTC_AF1
RTC_AF1
4 KB BKPSRAM
GPIO PORT A
GPIO PORT B
GPIO PORT C
GPIO PORT D
GPIO PORT E
GPIO PORT F
GPIO PORT G
GPIO PORT H
GPIO PORT I
PA[15:0]
PB[15:0]
PD[15:0]
PE[15:0]
PF[15:0]
PG[15:0]
PH[15:0]
PI[11:0]

Exercise: How to Use GPIO?

GPIO Demo



- From now on, there will be no text code to prevent you from copy-and-pasting.
  - See the images from this slide and type the code on your own.
  - Make a new project, type, build, debug(upload) and run the code.
  - After running the program, press the on-board joystick, as you wish.

- See the action on-board LEDs when you push the center button.

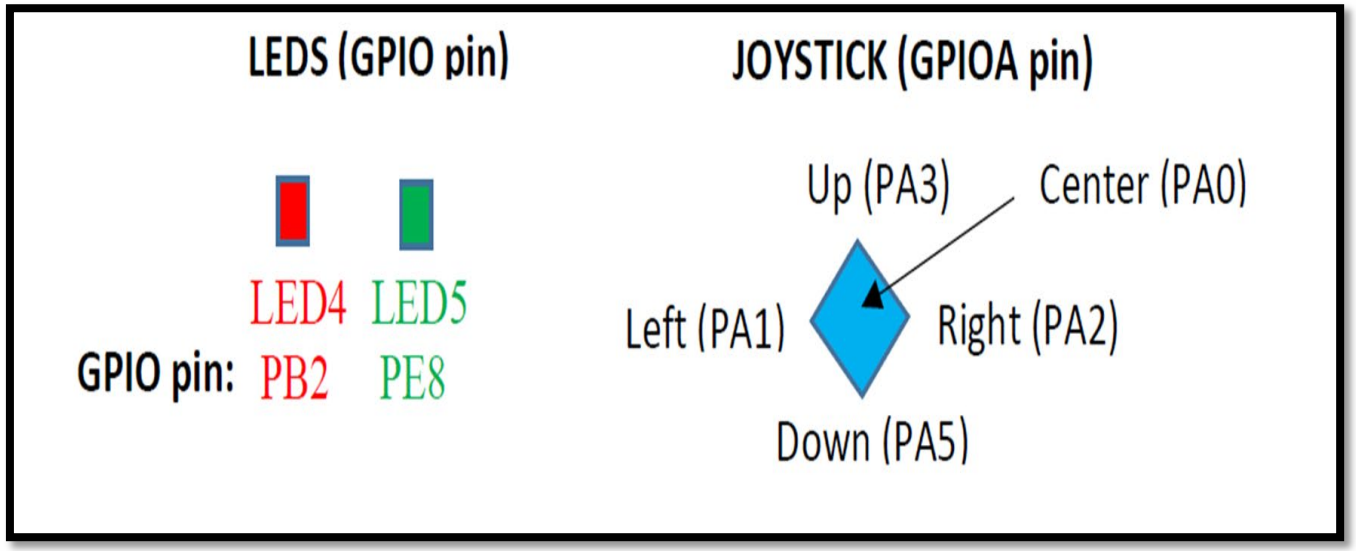
JOYSTICK Center:
Both LEDs turn OFF and remain OFF.

JOYSTICK Up:
1. LED4 ON
2. LED4 OFF

JOYSTICK Down:
1. LED5 ON
2. LED5 OFF

JOYSTICK Left:
1. LED5 ON
2. LED4 LED5 ON
3. Both OFF

JOYSTICK Right:
1. LED4 ON
2. LED4 LED5 ON
3. Both OFF



- *See program testing and submission below (20 MINUTES) .*

The program is to contain the following six C program modules:

1. An ***output handler***, which writes patterns to the LEDs.

2. An ***input handler***, which tests the JOYSTICK and returns some value that indicates which, if any, JOYSTICK button is pressed.

3. A ***system tick timer interrupt handler***, which is activated every one-half second. This routine should call the output handler, if the LEDs are to be changed.

4. A ***main program***, which executes in a continuous loop, calling the input handler every time through the loop.

5. The "startup" and "system initialization" files for the STM32L476G-Discovery board (inserted if you select "startup" under "device" in the "Manage Run-Time Environment" window when you create the project.)

6. The STM32L476xx microcontroller header file, stm32l476xx.h. (At top of main program, use ***#include "stm32l476xx.h"***

```c
#include <stm32l4xx.h>

typedef enum GPIO_Mode { GPIO_INPUT, GPIO_OUTPUT, GPIO_ALTERNATIVE, GPIO_ANALOG,
                         GPIO_INPUT_PULLUP, GPIO_INPUT_PULLDOWN = 0x8 } GPIO_Mode;

void ClockInit(void);
void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode);

int main(void)
{
  ClockInit();
  GPIO_Init(GPIOA, 0, GPIO_INPUT_PULLDOWN);
  GPIO_Init(GPIOA, 1, GPIO_INPUT_PULLDOWN);
  GPIO_Init(GPIOA, 2, GPIO_INPUT_PULLDOWN);
  GPIO_Init(GPIOA, 3, GPIO_INPUT_PULLDOWN);
  GPIO_Init(GPIOA, 5, GPIO_INPUT_PULLDOWN);
  GPIO_Init(GPIOB, 2, GPIO_OUTPUT);
  GPIO_Init(GPIOE, 8, GPIO_OUTPUT);

  while (1)
  {
    if (GPIOA->IDR & GPIO_IDR_ID1) GPIOB->BSRR = GPIO_BSRR_BS2;
    else if (GPIOA->IDR & GPIO_IDR_ID2) GPIOE->BSRR = GPIO_BSRR_BS8;
    else if (GPIOA->IDR & GPIO_IDR_ID3) GPIOB->BSRR = GPIO_BSRR_BR2;
    else if (GPIOA->IDR & GPIO_IDR_ID5) GPIOE->BSRR = GPIO_BSRR_BR8;
    else if (GPIOA->IDR & GPIO_IDR_ID0)
    {
      GPIOB->BSRR = GPIO_BSRR_BR2;
      GPIOE->BSRR = GPIO_BSRR_BR8;
    }
  }
}
```

```
34  void ClockInit(void)
35  {
36    FLASH->ACR |= FLASH_ACR_LATENCY_4WS;
37
38    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | (20 << RCC_PLLCFGR_PLLN_Pos)
39                 | RCC_PLLCFGR_PLLM_0 | RCC_PLLCFGR_PLLSRC_HSI;
40
41    RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;
42
43    while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS) && (RCC->CR & RCC_CR_PLLRDY)
44          && (RCC->CR & RCC_CR_HSIRDY)));
45
46    RCC->CFGR = RCC_CFGR_SW_PLL;
47
48    RCC->CR &= ~RCC_CR_MSION;
49  }
50
51  void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode)
52  {
53    unsigned int modeIn32Bit = ((mode & 3) << (2 * pin));
54    unsigned int pullUpDown = ((mode >> 2) << (2 * pin));
55
56    RCC->AHB2ENR |= (1 << (((unsigned int)port - GPIOA_BASE) >> 10));
57
58    port->MODER |= modeIn32Bit;
59    port->MODER &= (modeIn32Bit | ~(3 << (2 * pin)));
60
61    port->PUPDR |= pullUpDown;
62    port->PUPDR &= (pullUpDown | ~(3 << (2 * pin)));
63  }
64
```

Clock Enable

6.4.17 AHB2 peripheral clock enable register (RCC_AHB2ENR)

Address offset: 0x4C

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access

Note: When the peripheral clock is not active, the peripheral registers read or write access is not supported.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNG EN	HASHE N	AESEN (1)
													rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	DCMIE N	ADCEN	OTGFS EN	Res.	Res.	Res.	GPIOIE N	GPIOH EN	GPIOG EN	GPIOF EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
	rw	rw	rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

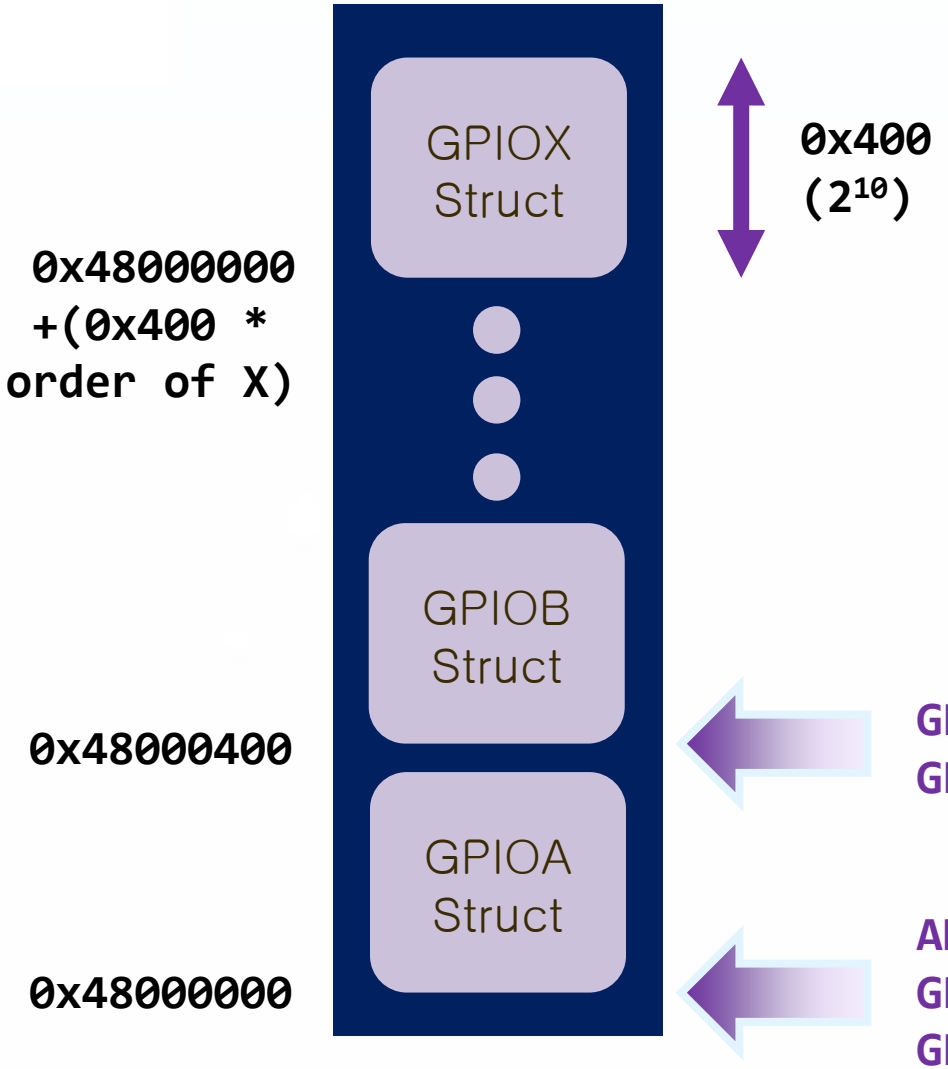
1. Available on STM32L42xxx, STM32L44xxx and STM32L46xxx devices only.

Please DON'T FORGET TO TURN ON THE CLOCK of peripherals before you use them! This is THE LAST TIME clock enabling is mentioned!

```
RCC->AHB2ENR |= (1 << <Port # you want>);
```

- Before you use any peripheral, you must enable the clock to it.
- The clock is enabled by RCC. There are so many peripherals, as well as their clock enable registers. Basically they are named after the bus where the peripherals are connected.
- To know which bus and register are connected to which peripheral, see the datasheet of STM32L4xx series.
- All GPIOs are connected to AHB and controlled by `AHB2ENR` register.

Clock Enable



But when you make a general initialization function, there is a problem that you need to know both the port address and the port number.

You may use your own macros, but there is a better way with predefined ones.

Before that, you should know the memory locations of GPIO structures.
They are aligned in a row, starting from GPIOA, and then GPIOB, and so on.

The size of each structure is 2^{10}, so the predefined pointer constant `GPIOB` is 2^{10} bigger than `GPIOA`.

Clock Enable

```
GPIOA = 0x48000000
GPIOB = 0x48000400
GPIOC = 0x48000800
GPIOX = 0x48000000 + 0x400 * # of X
```

```
GPIOA = 0
GPIOB = 1
GPIOC = 2
GPIOX = # of X
```

What we want is the bit number of each port.
To get the number of each port, we can use the formula below:

```
# of GPIOX = (GPIOX – 0x48000000) / 0x400
# of GPIOX = (GPIOX – GPIOA_BASE) >> 10
```

The two sentences are the same.
From above, the code to turn on the clock of the port of GPIO with the predefined GPIO macro pointer constant(`GPIO_TypeDef *port`) is:

```
RCC->AHB2ENR |= (1 << (((unsigned int)port - GPIOA_BASE) >> 10));
```

GPIO Mode

8.4.1 GPIO port mode register (GPIOx_MODER) (x =A to I)

Address offset:0x00

Reset value:

- 0xABFF FFFF (for port A)
- 0xFFFF FEBF (for port B)
- 0xFFFF FFFF (for ports C..G), I
- 0x0000 000F (for port H)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MODE[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

The point of this register you should remember:

1. Two bits for a pin
2. Default value is ‘1’, not ‘0’
3. You may need to both turn on and off the bits.

Each pin of GPIO has 4 modes.

- **(Digital) Input**
- **General Purpose(Digital) Output**
- **Alternative Function**
- **Analog(Default)**

Alternative function mode is used with special peripherals that use GPIO pins(ex. USART, Timer).

Analog mode is used with analog-related peripherals, or just to keep the wasted current to be low(the reason this mode is the default mode).

GPIO Mode

```
typedef enum GPIO_Mode { GPIO_INPUT, GPIO_OUTPUT, GPIO_ALTERNATIVE, GPIO_ANALOG,
                         GPIO_INPUT_PULLUP, GPIO_INPUT_PULLDOWN = 0x8
                       } GPIO_Mode;
```

To make things easier, the enumerator `GPIO_Mode` is declared.

```
          GPIO_INPUT = 0b0000
         GPIO_OUTPUT = 0b0001
    GPIO_ALTERNATIVE = 0b0010
         GPIO_ANALOG = 0b0011
   GPIO_INPUT_PULLUP = 0b0100
 GPIO_INPUT_PULLDOWN = 0b1000
```

There are something like pull-up or pull-down. Those will be explained later. They are inputs anyway, so the last 2 digits are '`0b00`' like a normal input.

In C, the exact type of the enumerator is compiler-dependent, so we should cast that value into unsigned int.

```
unsigned int modeIn32Bit
```

If you ignore the type-casting, you will have a really bad time. I warned you!

There are 3 steps to change the mode of the GPIO pin.
Assume that you want to make the mode of the pin 3 to output, and you don't know what the current state of `MODER` is.

GPIO Mode

1. **Get only the last 2 digits of 'modeIn32Bit'. To do that, AND it with 3(0b11).**
Then shift 'modeIn32Bit' left to where the bits we want are. Be aware that two bits are for a pin.

```
unsigned int modeIn32Bit = ((mode & 3) << (2 * pin));
```

With our example, the current value of '`modeIn32Bit`' is `0x40`.

2. **Turn on the bit that 'modeIn32Bit' has '1' in that position, by Bitwise OR.**

```
port->MODER |= modeIn32Bit;
```

From this point, you are sure that the bit 6 of `MODER` is '1'.

3. **Fill '1' in every bit in 'modeIn32Bit' except the two bits we are interested in.**
Then turn off the bit that 'modeIn32Bit' has '0' in that position, by Bitwise AND.

```
port->MODER &= (modeIn32Bit | ~(3 << (2 * pin)));
```

With our example, the mask you AND with `MODER` is `0xFFFF FF7F`.
From this point, you are sure that the bit 7 of `MODER` is '0'.

Now the value of `MODER` for pin 3 is surely '0b01', and that means the pin 3 is configurated as an output pin.

Pull-Up & Pull-Down

8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A to I)

Address offset: 0x0C

Reset value: 0x6400 0000 (for port A)

Reset value: 0x0000 0100 (for port B)

Reset value: 0x0000 0000 (for other ports)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPD15[1:0]		PUPD14[1:0]		PUPD13[1:0]		PUPD12[1:0]		PUPD11[1:0]		PUPD10[1:0]		PUPD9[1:0]		PUPD8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPD7[1:0]		PUPD6[1:0]		PUPD5[1:0]		PUPD4[1:0]		PUPD3[1:0]		PUPD2[1:0]		PUPD1[1:0]		PUPD0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **PUPD[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Like port mode register, the 2 bits decide the pull-up or pull-down behavior of a pin.

You already have the information of pull-up or pull-down, when you get the variable '`mode`' with the type of '`GPIO_Mode`' enumerator.

```
   GPIO_INPUT_PULLUP = 0b0100
 GPIO_INPUT_PULLDOWN = 0b1000
```

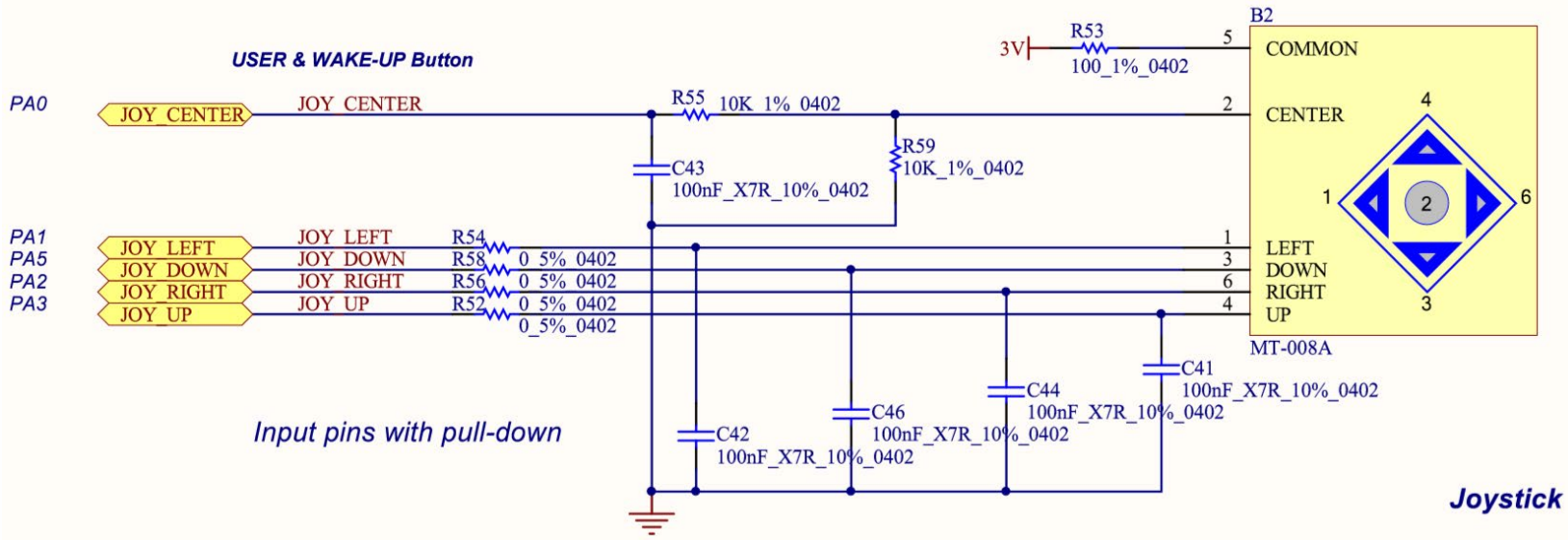
If you get the bit 2, bit 3 value and assign them to PUPDR, you can configure the appropriate pull-up / pull-down settings.

The way to set these bits are very similar to the mode setting.

```
unsigned int pullUpDown = ((mode >> 2) << (2 * pin));
port->PUPDR |= pullUpDown;
port->PUPDR &= (pullUpDown | ~(3 << (2 * pin)));
```

Pull-Up & Pull-Down

- Be aware, R53 is NOT a pull-up resistor. It is just a current-limiter, for when you use these as outputs and you accidently press the joystick.



This is the schematics of the on-board joystick. The common node of all internal switches are connected to V_{DD}(3V). All we need are pull-down resistors to each pin.
In `main()`, you can check those pins are initialized as `INPUT_PULLDOWN`.

What you are using is pull-down, so when you push the joystick, the input value will be '1'.

Input & Output

- `GPIOX->IDR & (1 << <pin#>)` is also okay. Use whatever you are convenient with.
- The reason you don't need Bitwise Shifting in condition checking, is that C regards all non-zero values to true.

8.4.5 GPIO port input data register (GPIOx_IDR) (x = A to I)

Address offset: 0x10

Reset value: 0x0000 XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A to I)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

When the pin is configured as input or output, then you can read or write the logical value from/to the pin. To read the input pin, you can use:

```
(variable) = (GPIOX->IDR & GPIO_IDR_ID<pin#>) >> <pin#>;
```

For example, when you want to read the pin 5 of GPIOB and store it to the variable x, the appropriate sentence is as below:

```
x = (GPIOB->IDR & GPIO_IDR_ID5) >> 5;
```

When you use the value as a logical value(true or false), then you don't need Bitwise Shifting.

```
if (GPIOB->IDR & GPIO_IDR_ID5) { … }
```

Input & Output

Similarly, you could think of setting the output of the pin as below:

```
GPIOX->ODR |= (1 << <pin#>);
GPIOX->ODR &= ~(1 << <pin#>);
```

If your program is so simple and do not use interrupts and thread scheduling, they are good enough. But one of your interrupt request handlers accesses to the same pin, it can cause a problem.

Let's dig into the first code above deeper. That code is the same as below:

```
GPIOX->ODR = (GPIOX->ODR | (1 << <pin#>));
```

It is composed of three stages: Read-Modify-Write.

1. Read

```
GPIOX->ODR = (GPIOX->ODR | (1 << <pin#>));
```

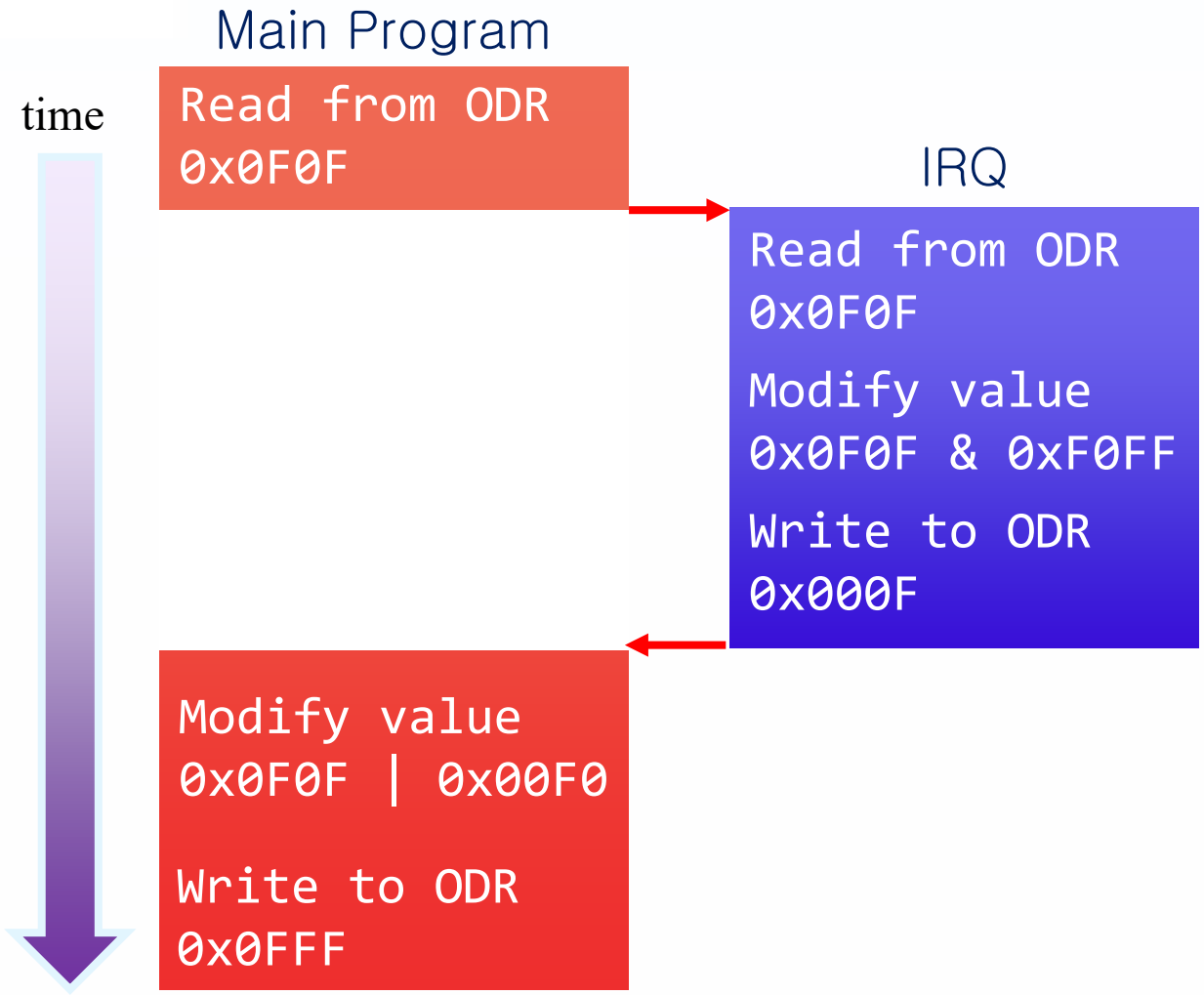
2. Modify

```
GPIOX->ODR = (GPIOX->ODR | (1 << <pin#>));
```

3. Write

```
GPIOX->ODR = (GPIOX->ODR | (1 << <pin#>));
```

Input & Output



Now, think about the situation that the IRQ and the main program access to the same ODR, and the interrupt is generated when the main program was reading ODR.

IRQ writes '0's to the bit #8 ~11.
Main Program writes '1's to the bit #4 ~ 7.
It seems there are no relations.

But when the control from IRQ returns to the main program, the bit #8 ~ 11 of the value being modified still remain '1'.

In conclusion, IRQ has no effect on ODR.
Considering the interrupts typically deals with more important tasks, it is a serious problem.

Atomic Set/Clear

- `GPIOX->BSRR = (1 << <pin#>)` or `GPIOX->BSRR = (1 << <pin# + 16>)` are also okay.

8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A to I)

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 **BR[15:0]:** Port x reset I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODx bit
1: Resets the corresponding ODx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BS[15:0]:** Port x set I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODx bit
1: Sets the corresponding ODx bit

To use it is very simple.

```
GPIOX->BSRR = GPIO_BSRR_BS<pin #>;
GPIOX->BSRR = GPIO_BSRR_BR<pin #>;
```

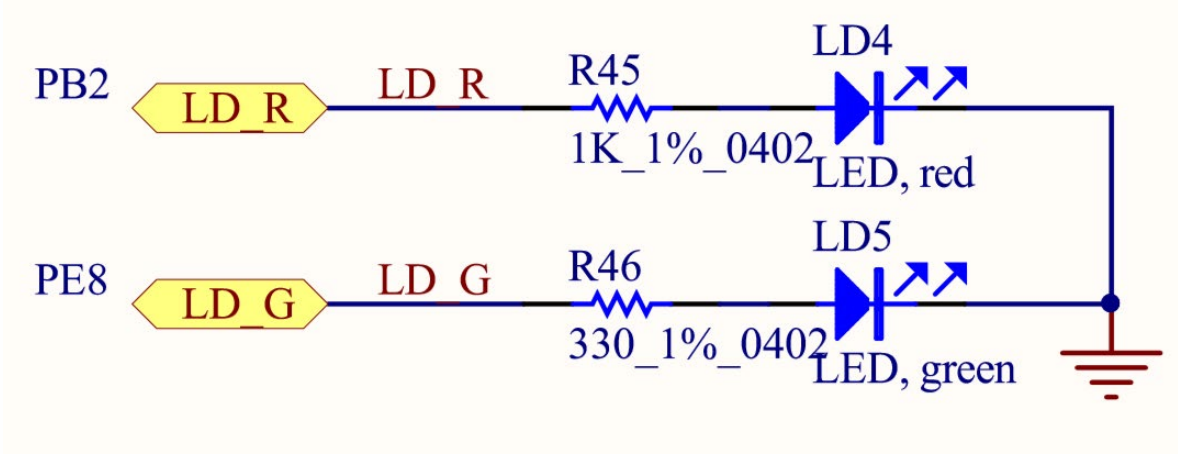
To prevent that, there is a special register to support atomic approach to ODR.

- When you write '1' to BS bit in BSRR, the corresponding output of the pin is set to 1.
- When you write '1' to BR bit in BSRR, the corresponding output of the pin is cleared to 0.
- When you write '0' to any of the bits, it has no effect at all.

With this, you can set or clear the output bits at once, without Read-Modify-Write.

Atomic Set/Clear

- Common-cathode means every diode's common node is its negative voltage node.



This is the schematics of on-board LEDs.

They are common-cathodes, so if you set the output to 1, you can turn on the LED.

In `main()`, if Left is pushed, the red LED is on, if Right is pushed, the green LED is on.

If Up is pushed, the red LED is off, if Down is pushed, the green LED is off.

If Center is pushed, all LEDs are off.

Limitations

GPIOs are not the super pins. You should be aware of the limitations of GPIOs.

- Input voltage must be bigger than -0.3V, smaller than 4.0V, except 5V tolerant pins.
  If the 5V tolerant pins are used for inputs with more than 4.0V, you can't use internal pull-up or pull-down resistors.

- Current in a pin is limited to -20~20mA.
  If a pin is supplying almost 20mA of current, the voltage might not be exactly 0 or V_{DD}.

- Total current into/out of all pins is limited to -100~100mA.
  (But if you use the voltage pins on-board, you can use the current up to 300mA.)
  (If the LED named LD3 lights up, it means your board is using the current over 300mA.)

- Rise and fall time of the output can be typically 25ns, regarding the load capacitance.

For more information about limitations, you should look up to the datasheet of STM32L476.

USART

USART

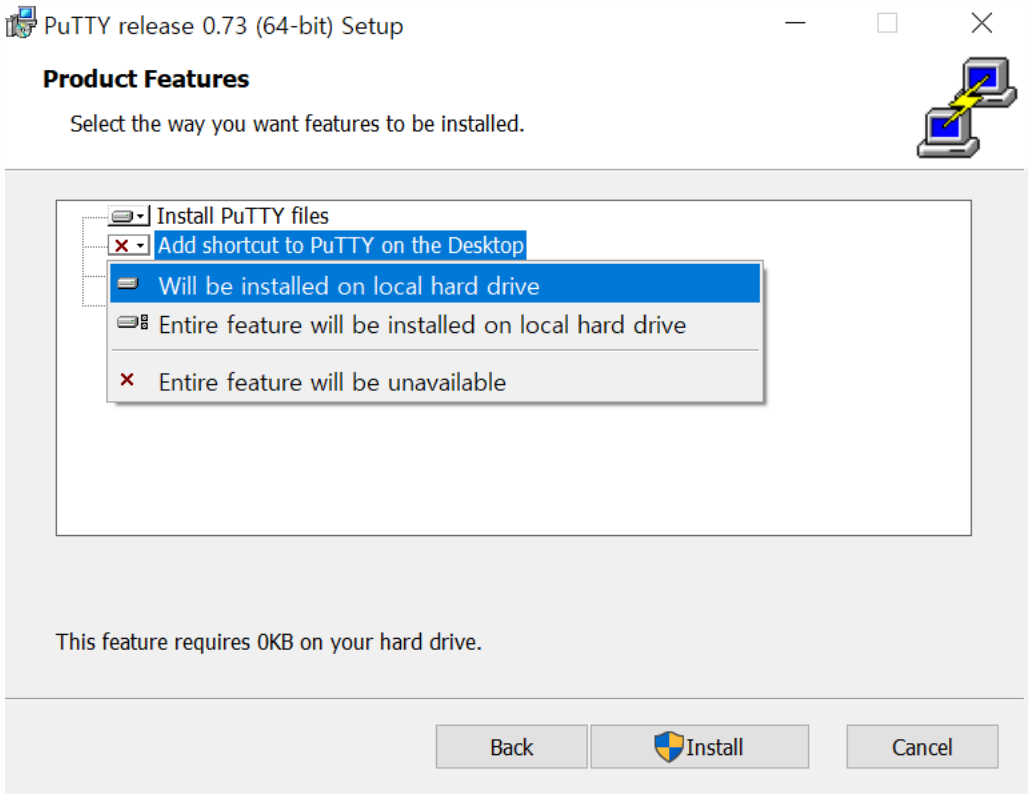
- USART stands for Universal Synchronous/Asynchronous Receiver/Transmitter.
- Simple but powerful serial communication.
- Minimum 2 wires are needed(in asynchronous mode).
- 1:1 communication, full-duplex(unless otherwise set to the other modes).
- 3 USART, 2 UART, 1 LPUART(Low Power UART) in STM32L4xx.
- Discovery board can communicate with a PC via USART2 using PD5, PD6.

How to Use It?

This time, you will make your discovery board to communicate with your PC.
Before that, you need a program called ‘terminal’, which is the interface between a PC and a device that uses COM port.
We will use ‘PuTTY’ this time.

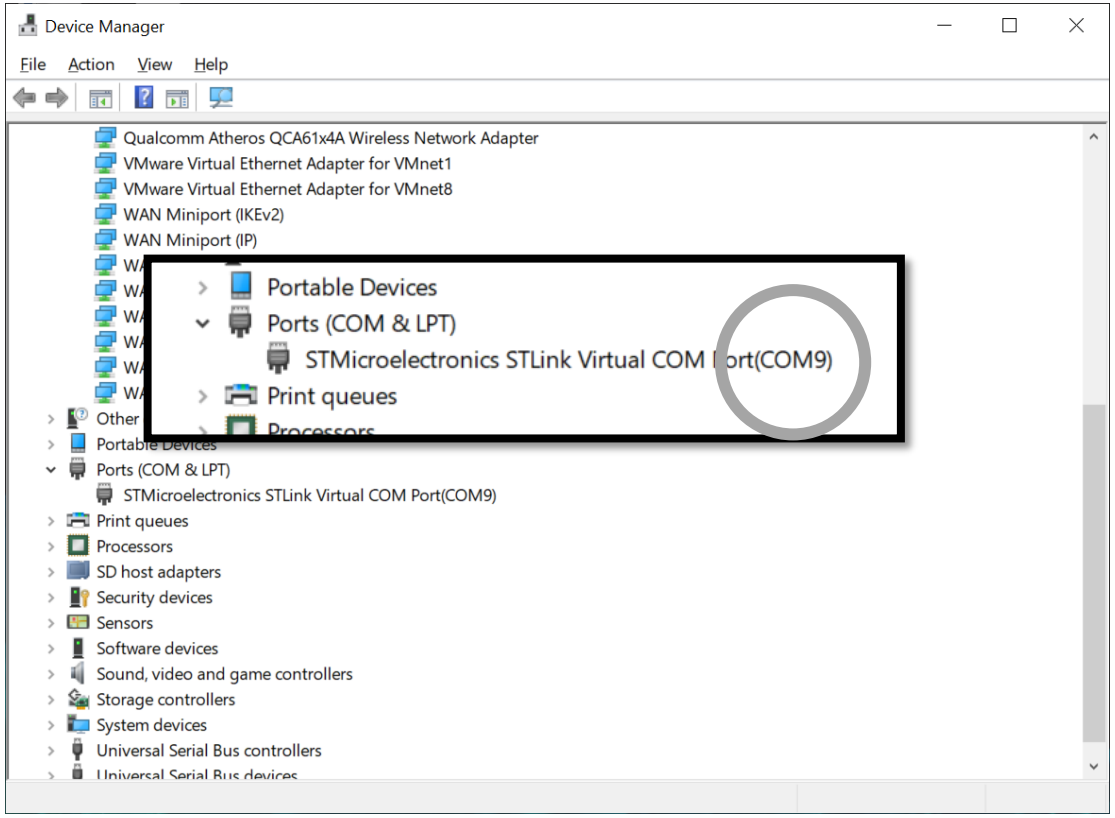
https://www.putty.org/

Download PuTTY and install it.
During installation, you may see this window.
I recommend you add shortcut to Desktop.



How to Use It?

During the installation, search ‘Device Manager’ at the taskbar of your Windows.
If you can’t find it, go to Windows Settings and search there.
If you really can’t find it, go to **Control Panel->Hardware and Sound -> Devices and Printers -> Device Manager**.



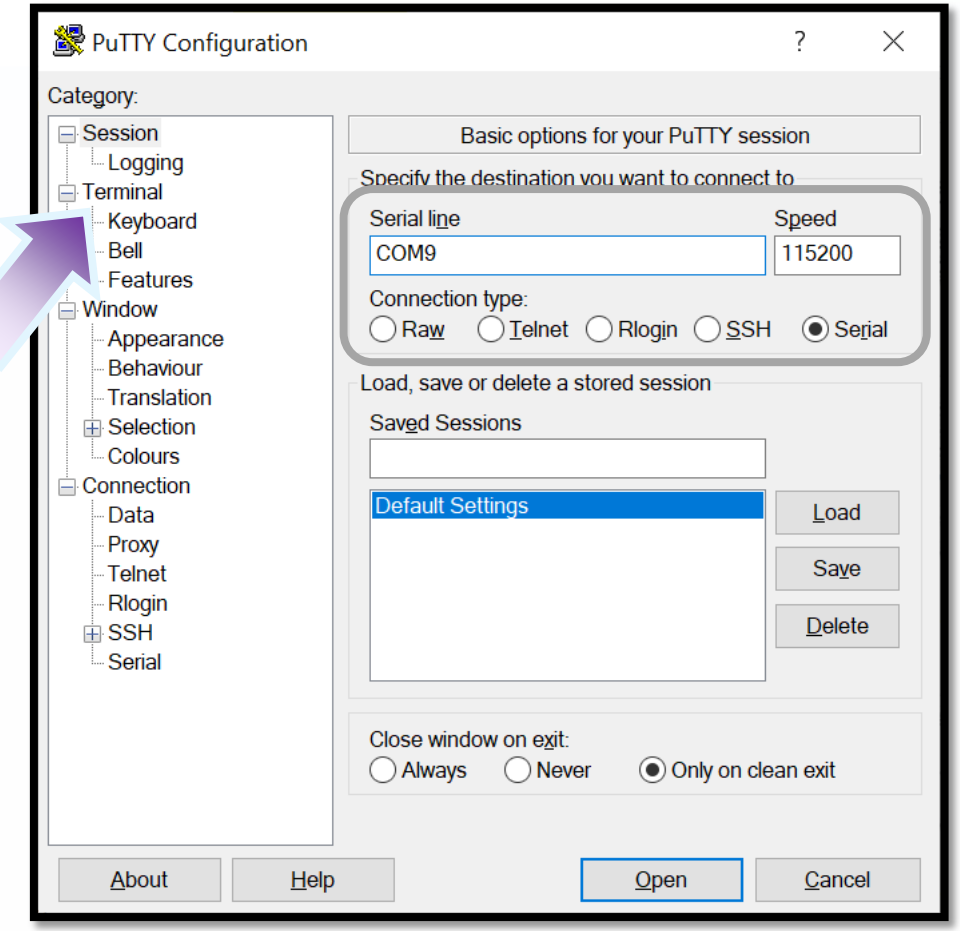
Go to Ports (COM & LPT).

Find ‘STMicroelectronics STLink Virtual COM Port’.

Check what port number is allocated to your board.

If it is not present, and ‘USB Serial Port’ is there instead, remember that number.

How to Use It?



Now open PuTTY.

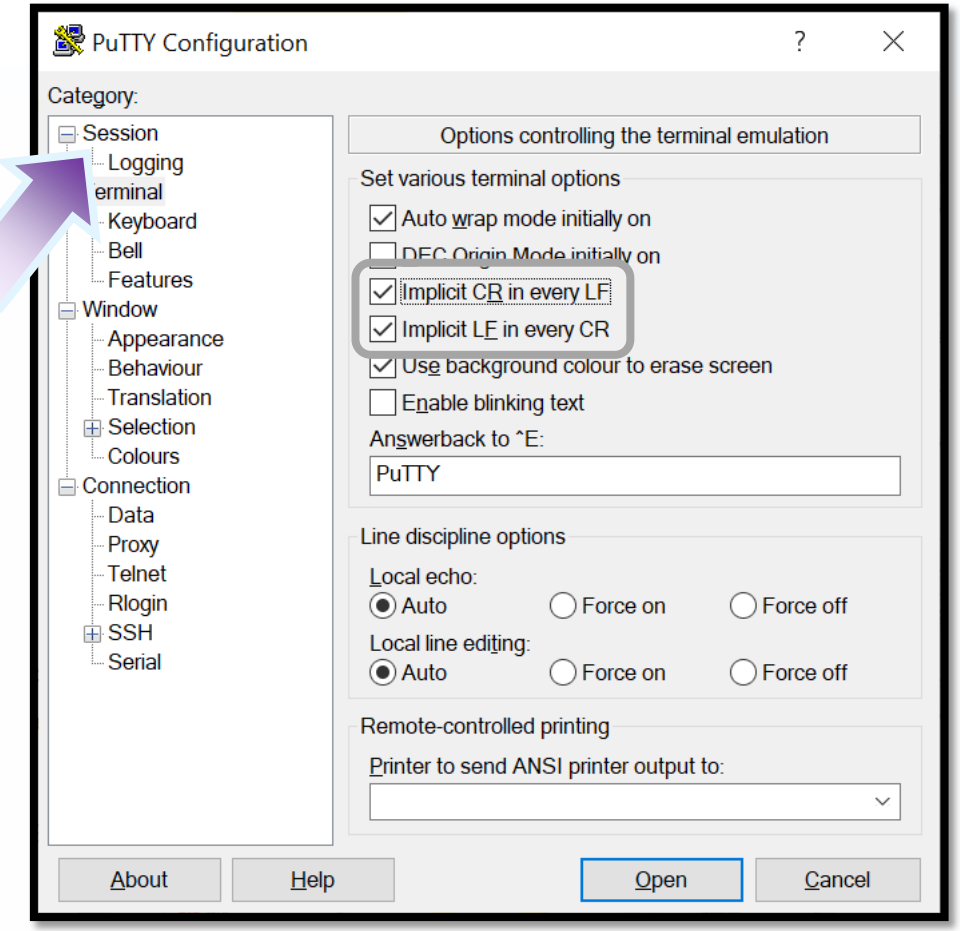
Select ‘Serial’ FIRST.

Type the COM port you checked.

Set the speed to 115200.

Then, go to ‘Terminal’.

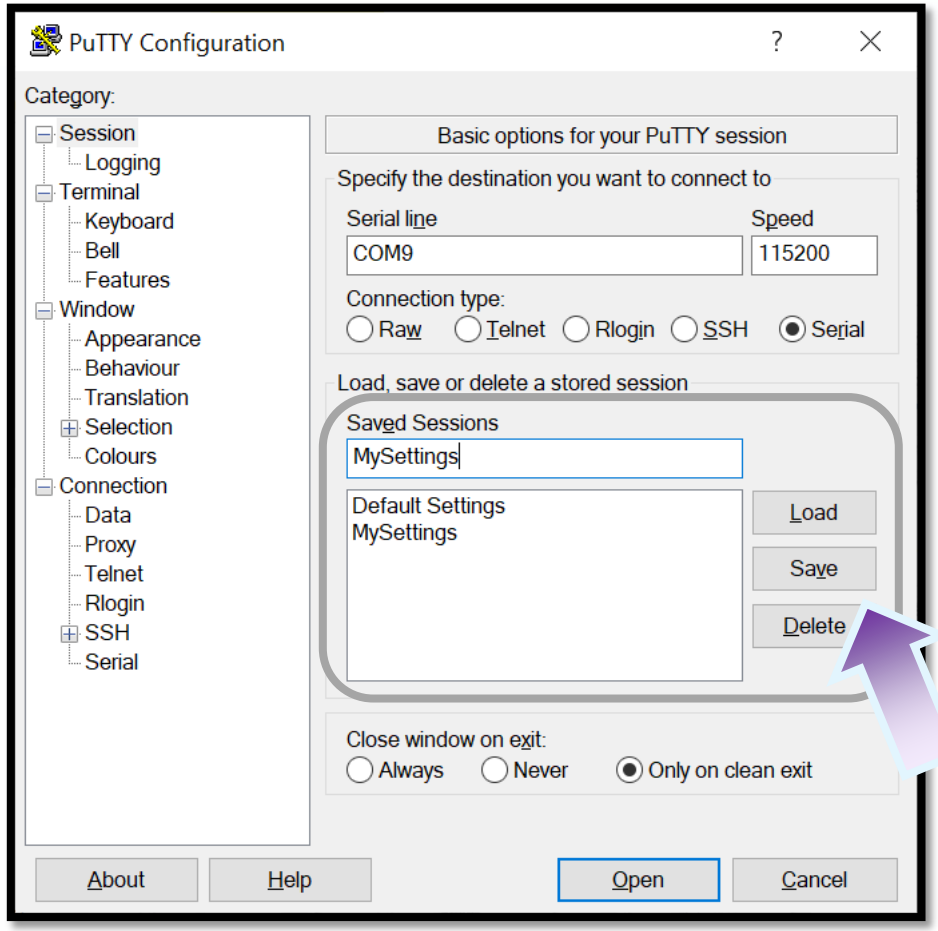
How to Use It?



There, check ‘Implicit CR in every LF’, ‘Implicit LF in every CR’.

Go back to ‘Session’.

How to Use It?

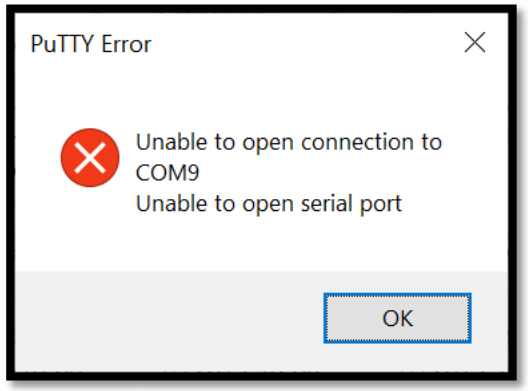


We will save these settings, since PuTTY loses all the settings when it starts.

Name your settings.
Click ‘Save’.

From now on, you can load your settings and click ‘Open’ to establish the session, or simply double click the name of your settings.

How to Use It?



If you encounter this message, please check again the COM port number of your board, and the connection between the board and your PC.

If you are still having trouble, and if you couldn’t find your device’s name in the Device Manager, you should have probably missed the driver.

Download the driver from the link below.
https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-utilities/stsw-link009.html

Install it **WITH YOUR BOARD DISCONNECTED**.

How to Use It?

If you are seeing this window, that’s good.
It doesn’t seem to do anything interesting, but this window shows the connection is established.

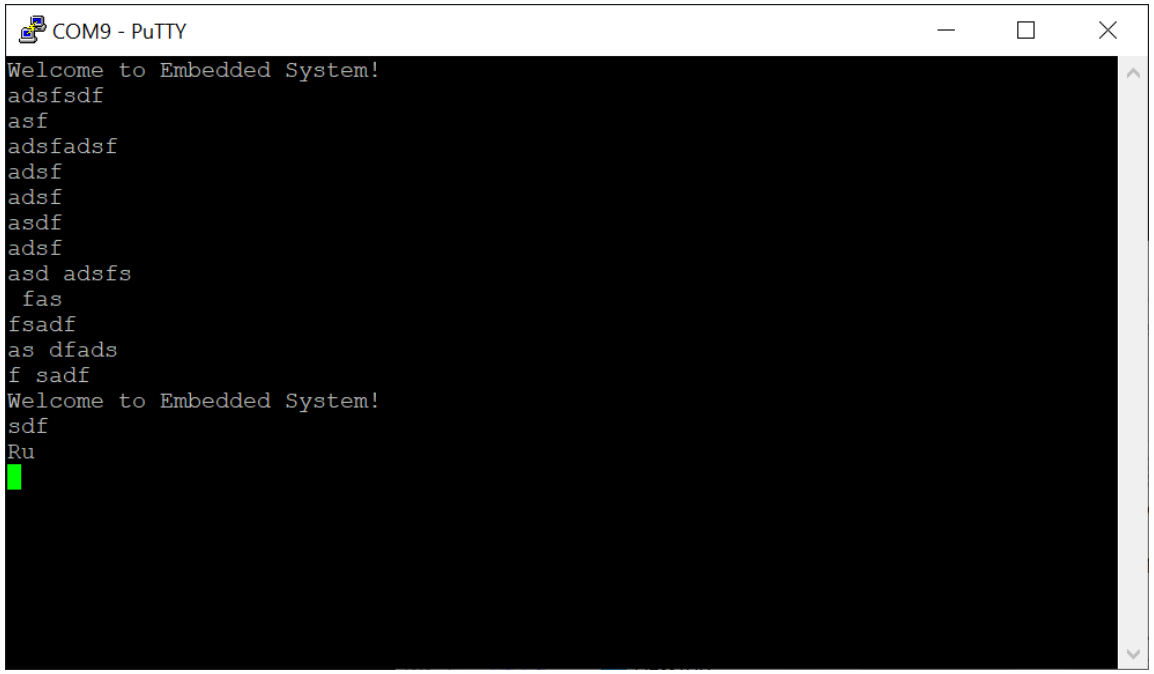
Now, let’s go to coding part.

```c
#include <stm32l4xx.h>

typedef enum GPIO_Mode { GPIO_INPUT, GPIO_OUTPUT, GPIO_ALTERNATIVE, GPIO_ANALOG,
                         GPIO_INPUT_PULLUP, GPIO_INPUT_PULLDOWN = 0x8 } GPIO_Mode;

void ClockInit(void);
void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode);
void USART2_Init(void);

char USART2_RX(void);
void USART2_TX(char character);
void USART2_TX_String(const char *string);

int main(void)
{
  ClockInit();
  USART2_Init();

  USART2_TX_String("Welcome to Embedded System!\n");

  while (1)
  {
    USART2_TX(USART2_RX());
  }
}
```

```
27  void ClockInit(void)
28  {
29    FLASH->ACR |= FLASH_ACR_LATENCY_4WS;
30
31    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | (20 << RCC_PLLCFGR_PLLN_Pos)
32                 | RCC_PLLCFGR_PLLM_0 | RCC_PLLCFGR_PLLSRC_HSI;
33
34    RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;
35
36    while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS) && (RCC->CR & RCC_CR_PLLRDY)
37          && (RCC->CR & RCC_CR_HSIRDY)));
38
39    RCC->CFGR = RCC_CFGR_SW_PLL;
40
41    RCC->CR &= ~RCC_CR_MSION;
42  }
43
44  void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode)
45  {
46    unsigned int modeIn32Bit = ((mode & 3) << (2 * pin));
47    unsigned int pullUpDown = ((modeIn32Bit >> 2) << (2 * pin));
48
49    RCC->AHB2ENR |= (1 << (((unsigned int)port - GPIOA_BASE) >> 10));
50
51    port->MODER |= modeIn32Bit;
52    port->MODER &= (modeIn32Bit | ~(3 << (2 * pin)));
53
54    port->PUPDR |= pullUpDown;
55    port->PUPDR &= (pullUpDown | ~(3 << (2 * pin)));
56  }
57
```

```
58  void USART2_Init(void)
59  {
60    RCC->CCIPR |= RCC_CCIPR_USART2SEL_1;
61    RCC->APB1ENR1 |= RCC_APB1ENR1_USART2EN;
62
63    GPIO_Init(GPIOD, 5, GPIO_ALTERNATIVE);
64    GPIO_Init(GPIOD, 6, GPIO_ALTERNATIVE);
65    GPIOD->AFR[0] |= (7 << GPIO_AFRL_AFSEL6_Pos) | (7 << GPIO_AFRL_AFSEL5_Pos);
66
67    USART2->BRR = 139;
68    USART2->CR3 |= USART_CR3_OVRDIS;
69    USART2->CR1 = USART_CR1_TE | USART_CR1_RE | USART_CR1_UE;
70  }
71
72  char USART2_RX(void)
73  {
74    while (!(USART2->ISR & USART_ISR_RXNE));
75    return USART2->RDR;
76  }
77
78  void USART2_TX(char character)
79  {
80    while (!(USART2->ISR & USART_ISR_TXE));
81    USART2->TDR = character;
82  }
83
84  void USART2_TX_String(const char *string)
85  {
86    while (*string != '\0') USART2_TX(*string++);
87  }
88
```

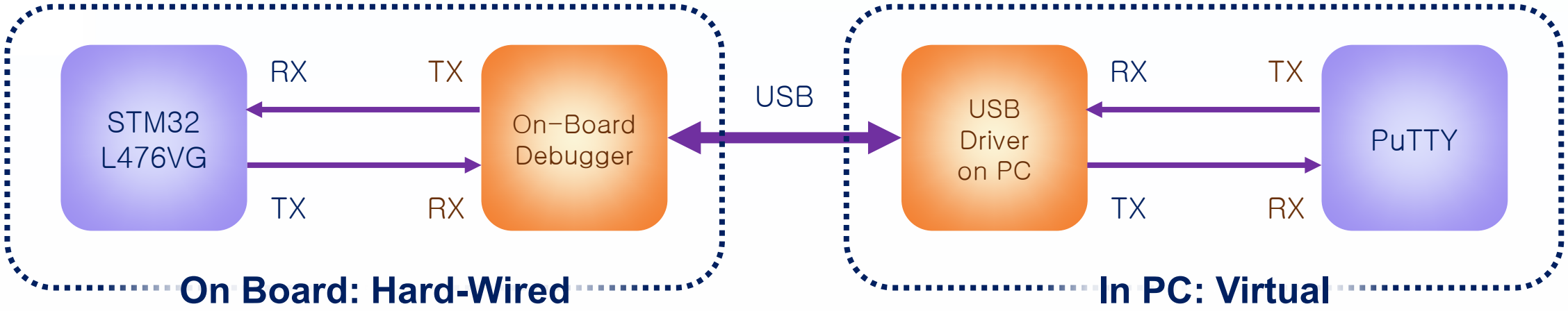
How to Use It?



Open the connection in PuTTY and Run your code.
If you missed the first message, you can see that by pushing the reset button on the board.

Type anything on PuTTY.
You may see what you typed.

What is happening?



In real world, to communicate with a PC, you should use USB, not 2-wire UART.
But USB is very hard to use, so there are some helper devices.

On the discovery board, on-board debugger acts like USB-to-TTL(UART) converter.
In the PC, the driver named USB Virtual COM let that USB device be a traditional COM(UART) port.

As a result, STM32L476VG thinks itself as communicating with PuTTY directly through UART.
This is only possible for USART2 using PD5, PD6 pins.

USART Registers

6.4.28 Peripherals independent clock configuration register (RCC_CCIPR)

Address: 0x88

Reset value: 0x0000 0000

Access: no wait states, word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DFSDM1SEL	SWPMI1SEL	ADCSEL[1:0]		CLK48SEL[1:0]		SAI2SEL[1:0]		SAI1SEL[1:0]		LPTIM2SEL[1:0]		LPTIM1SEL[1:0		I2C3SEL[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I2C2SEL[1:0]		I2C1SEL[1:0]		LPUART1SEL[1:0]		UART5SEL[1:0]		UART4SEL[1:0]		USART3SEL[1:0]		USART2SEL[1:0]		USART1SEL[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 3:2 **USART2SEL[1:0]**: USART2 clock source selection

This bit is set and cleared by software to select the USART2 clock source.

00: PCLK selected as USART2 clock

01: System clock (SYSCLK) selected as USART2 clock

10: HSI16 clock selected as USART2 clock

11: LSE clock selected as USART2 clock

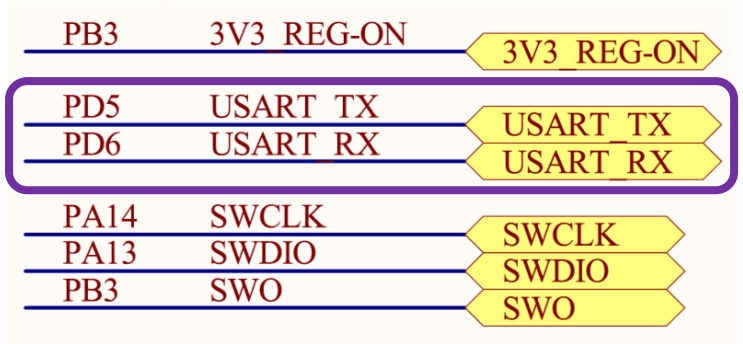
Some of time-critical peripherals get their own clock from specific clock sources.

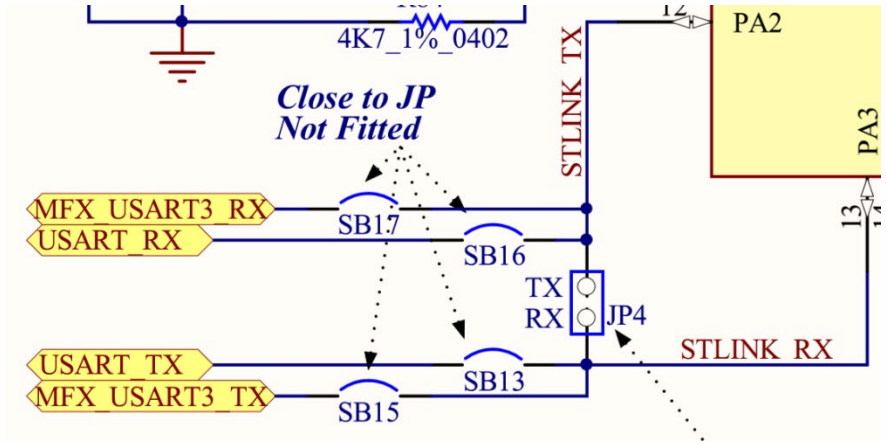
We will use HSI, only because it is sufficient to generate 115200 baud rate(and we did not configure APB).

You must also enable the clock, independent of clock source selection.

```
RCC->CCIPR |= RCC_CCIPR_USART2SEL_1;
```

USART Registers





According to the datasheet of the discovery board, PD5, and PD6 pins are used in communicating with on-board debugger.
So we need to configure that GPIOs.

We can think of it as a special peripheral(USART) is using those GPIO pins.
So those pins must be configured as alternative function mode.

```
GPIO_Init(GPIOD, 5, GPIO_ALTERNATIVE);
GPIO_Init(GPIOD, 6, GPIO_ALTERNATIVE);
```

USART Registers

Table 17. Alternate function AF0 to AF7[1] (continued)

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
		SYS_AF	TIM1/TIM2/ TIM5/TIM8/ LPTIM1	TIM1/TIM2/ TIM3/TIM4/ TIM5	TIM8	I2C1/I2C2/I2C3	SPI1/SPI2	SPI3/DFSDM	USART1/ USART2/ USART3
Port D	PD0	-	-	-	-	-	SPI2_NSS	DFSDM1_ DATIN7	-
	PD1	-	-	-	-	-	SPI2_SCK	DFSDM1_CKIN7	-
	PD2	-	-	TIM3_ETR	-	-	-	-	USART3_RTS_ DE
	PD3	-	-	-	-	-	SPI2_MISO	DFSDM1_ DATIN0	USART2_CTS
	PD5	-	-	-	-	-	-	-	USART2_TX
	PD6	-	-	-	-	-	-	DFSDM1_ DATIN1	USART2_RX
	PD7	-	-	-	-	-	-	DFSDM1_CKIN1	USART2_CK
	PD8	-	-	-	-	-	-	-	USART3_TX
	PD9	-	-	-	-	-	-	-	USART3_RX
	PD10	-	-	-	-	-	-	-	USART3_CK
	PD11	-	-	-	-	-	-	-	USART3_CTS
	PD12	-	-	TIM4_CH1	-	-	-	-	USART3_RTS_ DE
	PD13	-	-	TIM4_CH2	-	-	-	-	-
	PD14	-	-	TIM4_CH3	-	-	-	-	-
	PD15	-	-	TIM4_CH4	-	-	-	-	-

The alternative functions of a GPIO are not unique. There can be 16 options.

Again, according to the datasheet of STM32L476, the alternative function 7 for PD5 and PD6 is for GPIO2.

So in `AFR[0]` register, we will set the appropriate alternative function number 7.

If the pin number is bigger than 7, you must use `AFR[1]`.

```
GPIOD->AFR[0] |= (7 << GPIO_AFRL_AFSEL6_Pos) | (7 << GPIO_AFRL_AFSEL5_Pos);
```

'L' after AFR means `AFR[0]`. If you want to change `AFR[1]`, use `GPIOX_AFRH_....`

USART Registers

40.8.4 Baud rate register (USART_BRR)

This register can only be written when the USART is disabled (UE=0). It may be automatically updated by hardware in auto baud rate detection mode.

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
USART2->BRR = 139;
```

It should be bigger than 16.

If it isn't, find a faster clock source for the USART.

There is a register for Baud rate calculation. This register must be configured before enabling the USART.

The Baud rate from this register is calculated with the following formula:

Baud rate = clock freq. / BRR value

To obtain the BRR value, use this formula:

BRR value = clock freq. / Baud rate

We will use 16MHz HSI clock with 115200 Baud rate, the appropriate BRR value is 138.888… ≒ 139.

USART Registers

40.8.3 Control register 3 (USART_CR3)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TCBGTIE	UCESM	WUFIE	WUS1	WUS0	SCARCNT2	SCARCNT1	SCARCNT0	Res.
							rw	rw	rw	rw	rw	rw	rw	rw	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DEP	DEM	DDRE	OVRDIS	ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
USART2->CR3 |= USART_CR3_OVRDIS;
```

It is a rare case for the small program like what we are talking about.

But even the slight change of the program can make a chance for the CPU to miss the incoming data.
That situation is called ‘Data Overrun’ or ‘Buffer Overrun’.

In that case you must deal with it or ignore it by setting `OVRDIS` bit in `CR3` register.
Otherwise the receiver will stop working.

USART Registers

40.8.1 Control register 1 (USART_CR1)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEI	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

After configuring every other registers in USART, you must enable the receiver, the transmitter, and USART itself.

Those can be done at once.

```
USART2->CR1 = USART_CR1_TE | USART_CR1_RE | USART_CR1_UE;
```

Basic Way to Communicate

40.8.8 Interrupt and status register (USART_ISR)

Address offset: 0x1C

Reset value: 0x0200 00C0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	TCBGT	Res.	Res.	REACK	TEACK	WUF	RWU	SBKF	CMF	BUSY
						r			r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABRF	ABRE	Res.	EOBF	RTOF	CTS	CTSIF	LBDF	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
r	r		r	r	r	r	r	r	r	r	r	r	r	r	r

- TC(Transfer Complete) bit is set when the data is fully out of the device. On the other hand, TXE(Transmit Data Register Empty) bit is set when the buffer hands its content to the shift register, which transfer the data in serial. You can infer that using TC bit is slower than using TXE. TC is used for half-duplex mode, where the device has to know whether the communication line is fully idle or not.

Typically USART transfers the data, byte-by-byte, and its speed is much slower than the CPU.

To know the byte is ready to be read, check RXNE bit.
It is set when USART receives a bit and the software hasn't read the value yet.

To know the buffer is ready to be written, check TXE bit, NOT TC BIT!

It is set when the data is being sent, and the buffer is ready to be written.

Basic Way to Communicate

40.8.10 Receive data register (USART_RDR)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	RDR[8:0]								
							r	r	r	r	r	r	r	r	r

40.8.11 Transmit data register (USART_TDR)

Address offset: 0x28

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TDR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

The received data is in RDR, and you should put the data to be transmitted in TDR.

With these registers, you can make the receive and transmit functions.

```
char USART2_RX(void) {
    // Wait for the receiving be complete
    while (!(USART2->ISR & USART_ISR_RXNE));
    return USART2->RDR;
}

void USART2_TX(char character) {
    // Wait for the ongoing transmitting
    while (!(USART2->ISR & USART_ISR_TXE));
    USART2->TDR = character;
}
```

Basic Way to Communicate

With already existing transmitting function ‘USART2_TX()’, you can make a function that transmits a string.

```
void USART2_TX_String(char *string) {
    // Repeatedly call ‘USART2_TX()’ until the string ends
    // ‘string’ indicates the next character of the string
    // after calling ‘USART2_TX()’
    while (*string != ‘\0’) USART2_TX(*string++);
}
```

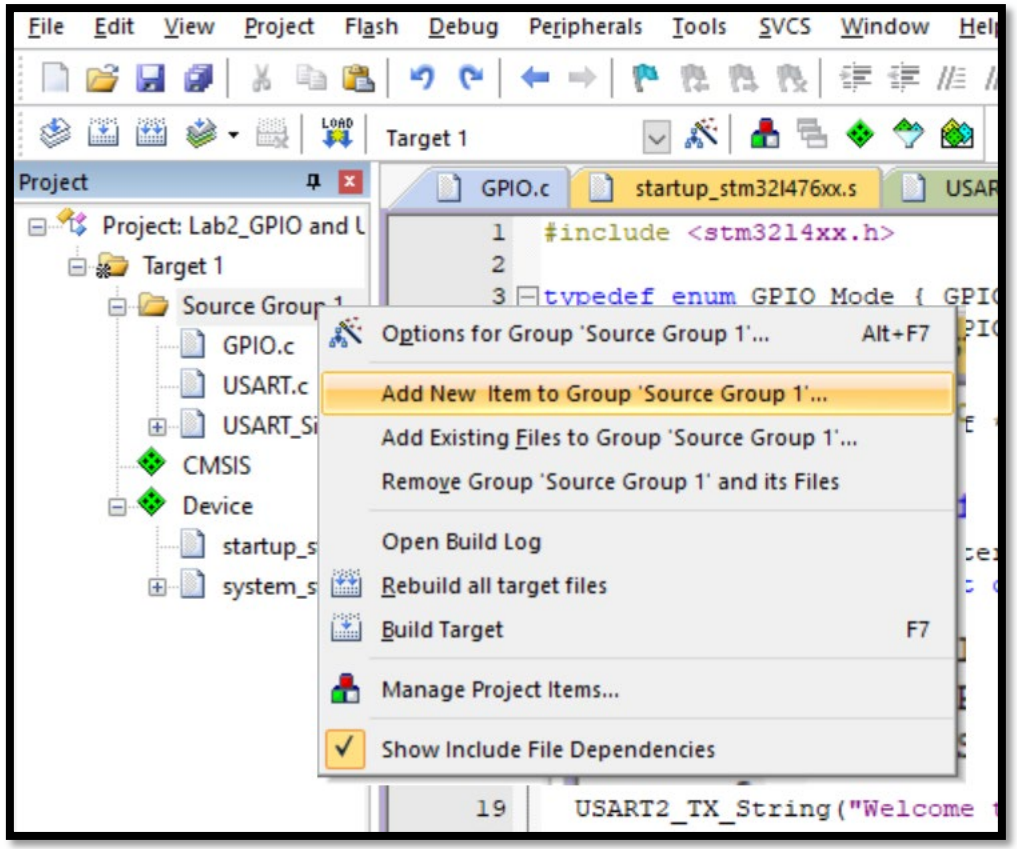
Do It Yourself!

Example Code Review

You've come so far.
Now you can add a comment on the example code which we have used since Lab0.
Take a look at the code again and comment yourself what those uncommented sentences do.

```
1  #include <stm32l4xx.h>
2
3  int main(void)
4  {
5    RCC->AHB2ENR = RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOEEN;
6    GPIOB->MODER &= ~GPIO_MODER_MODE2_1;
7    GPIOE->MODER &= ~GPIO_MODER_MODE8_1;
8
9    while (1)
10   {
11     GPIOB->BSRR = GPIO_BSRR_BS2;
12     GPIOE->BSRR = GPIO_BSRR_BS8;
13
14     for (int i = 0; i < 1000000; i++);
15
16     GPIOB->BSRR = GPIO_BSRR_BR2;
17     GPIOE->BSRR = GPIO_BSRR_BR8;
18
19     for (int i = 0; i < 1000000; i++);
20   }
21 }
22
```

Make the Libraries

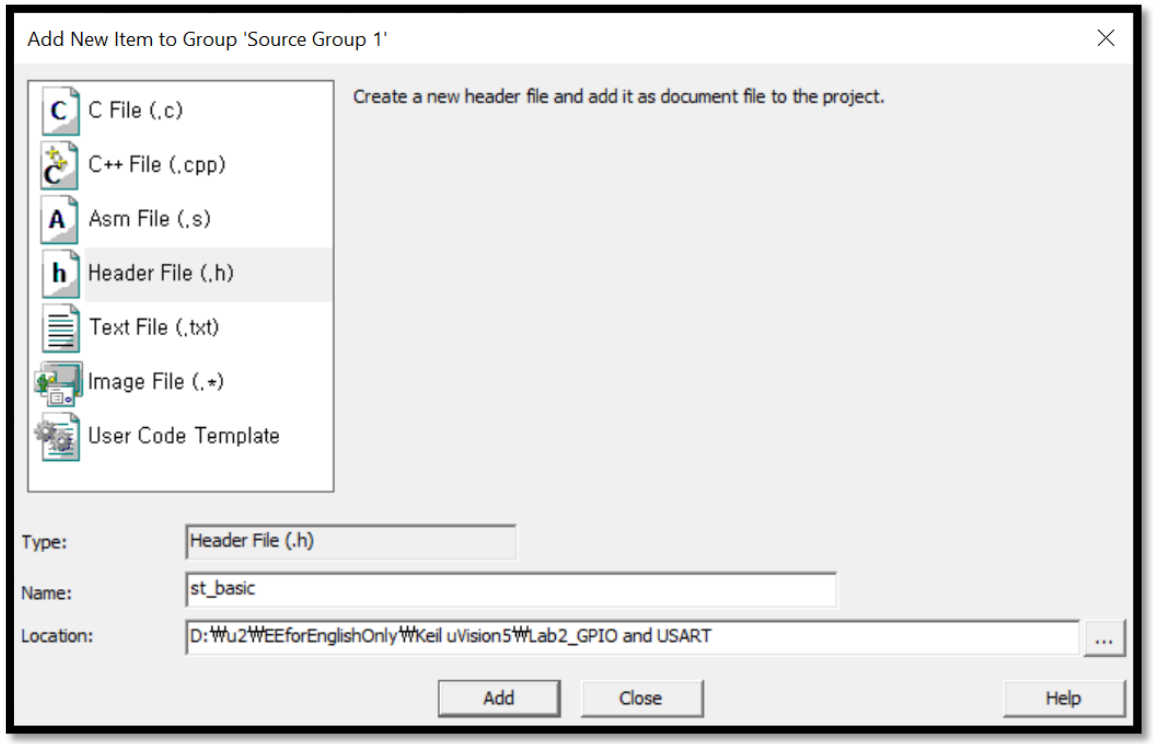


At this point, you may feel the code is already too long to easily understand the entire code.
Also, It became very hard to copy and paste the necessary codes like `Delay()` and `GPIO_Init()`, every time you need.

It is now obvious that you need your own library files.
After you write your own header files, all you have to do is to copy and paste the header file in your new project and `#include` that header file in your code.

To make your header file, right click the ‘Source Group 1’ folder and click ‘Add New item to Group ‘Source Group 1’…’

Make the Libraries



Select Header File and name it properly.

Unlike the project or source file names, it is good to name it properly.
You will use this library, type its name in your source code by `#include`, and modify until you satisfy.

If your header's name is too long and complicated or just simply contains too much capital letters, you will regret it when you include it in your new source.

Click 'Add'.

Make the Libraries

```
#ifndef _ST_BASIC_
#define _ST_BASIC_

#ifndef STM32L476xx
#define STM32L476xx
#endif
#include <stm32l4xx.h>

typedef enum GPIO_Mode { GPIO_INPUT, GPIO_OUTPUT, GPIO_ALTERNATIVE, GPIO_ANALOG,
                         GPIO_INPUT_PULLUP, GPIO_INPUT_PULLDOWN = 0x8 } GPIO_Mode;

void ClockInit(void);
void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode);
void USART2_Init(void);
void Delay(unsigned int duration);
char USART2_RX(void);
void USART2_TX(char character);
void USART2_TX_String(const char *string);

#endif
```

Then, move your macro constants, enumerators, structs, type definitions, and function declarations.
Here, no global variables and function definitions are.
No need to write down IRQ handlers.

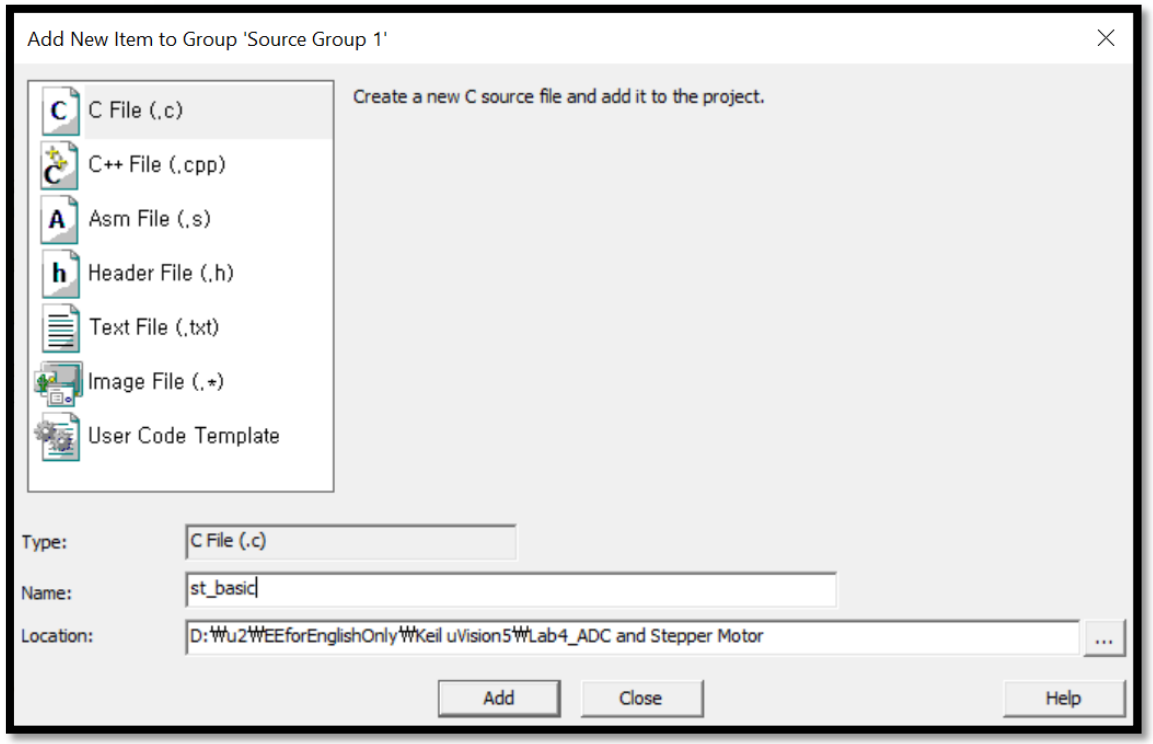
First, use `#ifndef` to prevent your header is included more than once.

```
#ifndef _ST_BASIC_
#define _ST_BASIC_
… <your code> …
#endif
```

Next, to prevent the IDE from displaying error messages when writing your code, define ‘STM32L476xx’ with `#ifndef` directive, and include <stm32l4xx.h> here.

```
#ifndef STM32L476xx
#define STM32L476xx
#endif
#include <stm32l4xx.h>
```

Make the Libraries



This time, you have to make another .c file of the header.

This file includes the global variables and function definitions that must be compiled only once.

Name it properly and Click ‘Add’.

Make the Libraries

```c
#include "st_basic.h"

unsigned int sysMillis = 0;

void ClockInit(void)
{
  FLASH->ACR |= FLASH_ACR_LATENCY_4WS;

  RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | (20 << RCC_PLLCFGR_PLLN_Pos)
               | RCC_PLLCFGR_PLLM_0 | RCC_PLLCFGR_PLLSRC_HSI;

  RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;

  while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS) && (RCC->CR & RCC_CR_PLLRDY)
        && (RCC->CR & RCC_CR_HSIRDY)));

  RCC->CFGR = RCC_CFGR_SW_PLL;

  RCC->CR &= ~RCC_CR_MSION;

  SysTick->LOAD = SysTick->CALIB & SysTick_LOAD_RELOAD_Msk;
  SysTick->CTRL = SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;
}

void GPIO_Init(GPIO_TypeDef *port, unsigned int pin, GPIO_Mode mode)
{
  unsigned int modeIn32Bit = ((mode & 3) << (2 * pin));
  unsigned int pullUpDown = ((mode >> 2) << (2 * pin));

  RCC->AHB2ENR |= (1 << (((unsigned int)port - GPIOA_BASE) >> 10));

  port->MODER |= modeIn32Bit;
  port->MODER &= (modeIn32Bit | ~(3 << (2 * pin)));

  port->PUPDR |= pullUpDown;
  port->PUPDR &= (pullUpDown | ~(3 << (2 * pin)));
}
```

First, `#include` the header file.
Then write the global variables and function definitions.

Here, you also need to implement the IRQ handlers.

Make the Libraries

```
1  #include "st_basic.h"
2
3  int main(void)
4  {
5    ClockInit();
6    USART2_Init();
7
8    USART2_TX_String("Welcome to Embedded System!\n");
9
10   while (1)
11   {
12     USART2_TX(USART2_RX());
13   }
14 }
15
```

Thanks to the library, your source code is now much shorter and easy to understand.

Just be careful, when you are including your own file in the project folder, the filename is wrapped with double quotation marks, not angle brackets.

Button Input to USART

Task 1

You've learned how to read the input of the GPIO.
You've also learned how to transmit the data to the other device, via USART.

This time, get the input from the on-board joystick, and send the direction information to the PC.

USART to LED

Task 2

You've learned how to receive the data from the other device, via USART.
You've also learned how to set the output of the GPIO.

This time, get the directional information from the PC and show it as a state of LEDs.
Use 'w', 'a', 's', 'd', 'space', for Up, Left, Down, Right, Center, for your first try.

If this is too easy for you, then try sending the string "Up", "Left", "Down", "Right", "Center".

THANK YOU