Queen Mary
University of London
Science and Engineering

School of Electronic Engineering and Computer Science
QMUL-BUPT Joint Programme

EBU6475 Microprocessor System Design
EBU5476 Microprocessors for Embedded Computing

General Purpose Input Output

References:
Chapter 2, Embedded Systems Fundamentals
Chapter 10.4, The Definitive Guide to ARM®

arm
University Program Education Kits

Last updated: 15 April, 2020

What have we learned so far?

- Embedded systems and Computer design
- Cortex-M4 core
- C as Implemented in Assembly Language
- Interrupts

This week we will talk about **GPIOs** and **Timers**

Overview

1. Input/Output Strategy of microprocessors
    a. Memory-mapped I/O
    b. Port-mapped I/O
2. General Purpose Input Output (GPIO)
    a. Concept
    b. Pins & Ports
    c. Electric parameters
3. Controlling the GPIO
    a. Control registers
    b. CMSIS
    c. C-Code
4. Examples
    a. Example 1
    b. Example 2

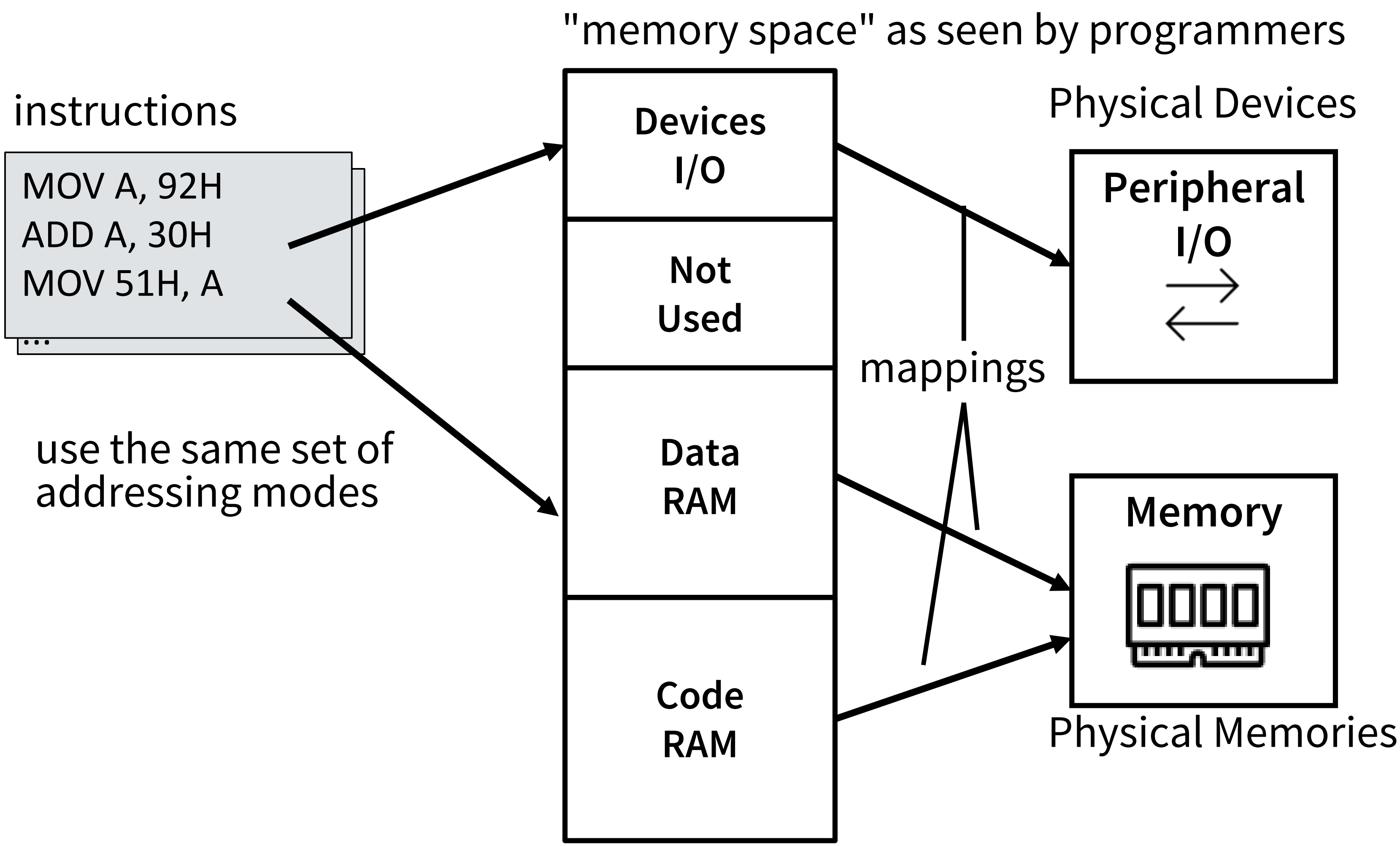
Input/Output Strategy of microprocessors

1. Input/Output Strategy of microprocessors
   a. Memory-mapped I/O
   b. Port-mapped I/O

I/O Styles in Microprocessors

- Memory Mapped I/O (MMIO)
  - Main memory and peripheral I/O devices map into one single address space
  - e.g. Intel MCS-51 (commonly known as 8051), ARM
- Port Mapped I/O (PMIO)
  - Peripherals have separate address space from main memory
  - Have special CPU instructions for I/O
    - like IN and OUT

Memory-Mapped I/O



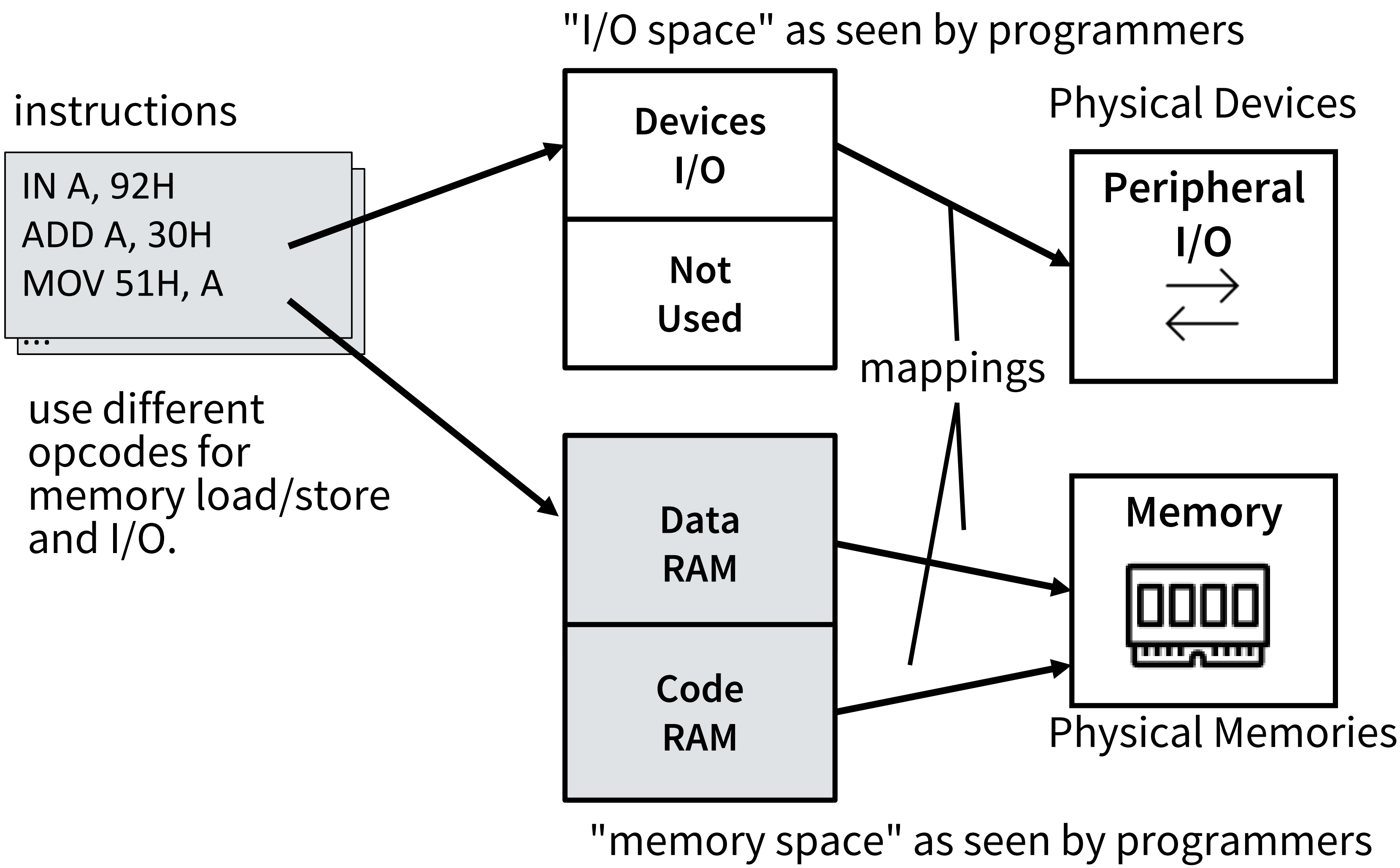
Memory-Mapped I/O: Pros & Cons

No extra external circuitry required for I/O access – simpler & cheaper.

Every instruction which can access memory can be used to manipulate an I/O device.

Uses up main memory space for peripherals

Port-Mapped I/O



use different opcodes for memory load/store and I/O.

Port-Mapped I/O: Pros & Cons

Separate I/O and memory access

Allows full memory space to be used for RAM

Obvious to see when I/O occurs in program code because of special I/O functions

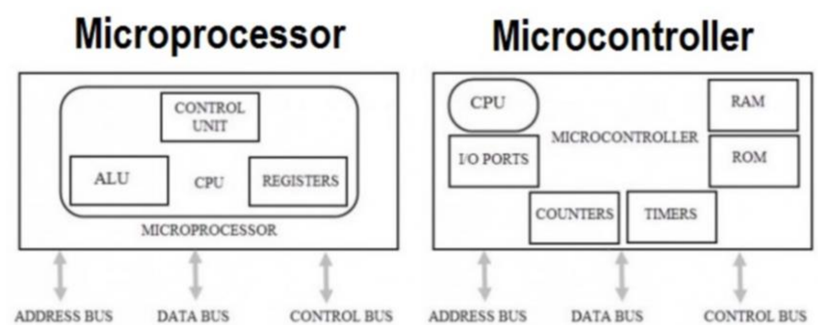
Introduces complexity to internal circuits

Special I/O functions harder to support for higher level language compilers

More instructions are required to accompolish the same task, e.g. test one bit on I/O

Input / Output Pins & Ports

- Most of embedded microprocessors have I/O memory-mapped
- One key feature of a microcontroller is the versatility built into the input/output (I/O) circuits.
  - Microprocessor designs, on the other hand, must add additional chips to interface with external circuitry.
- The function a pin performs can be easily programmed.

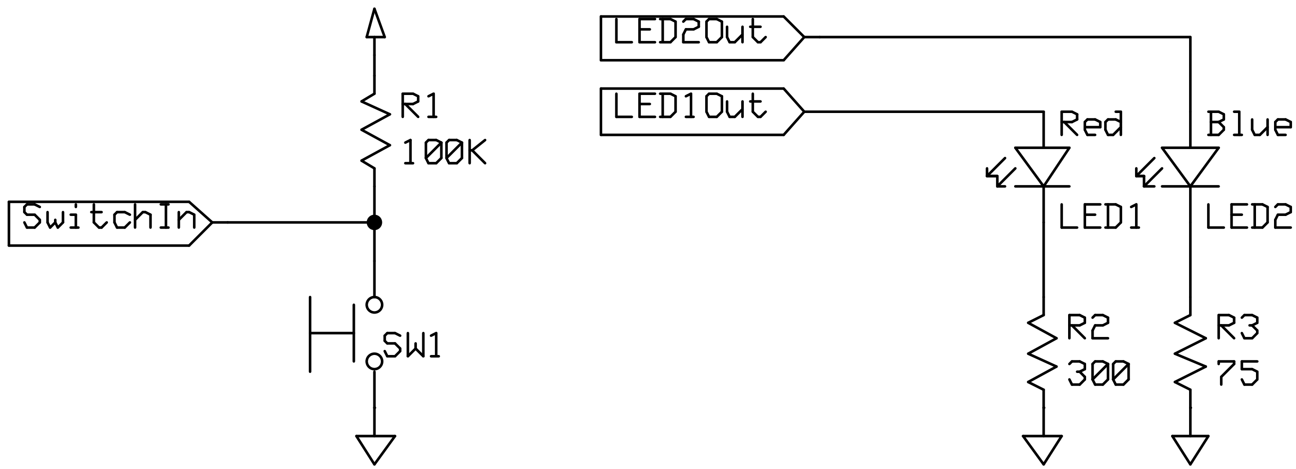


Microprocessor vs Microcontroller by EEEPROJECT.COM

General Purpose Input Output (GPIO)

2. General Purpose Input Output (GPIO)
    a. Concept
    b. Pins & Ports
    c. Electric parameters

GPIO Basic Concept



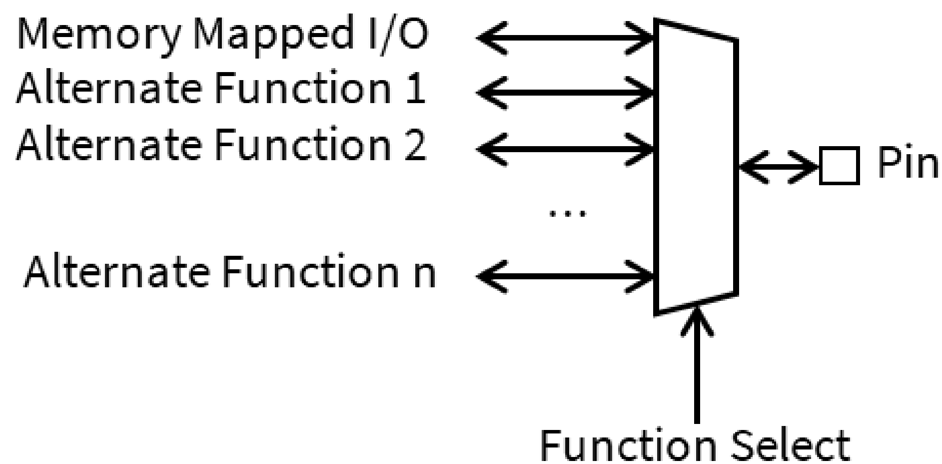
- GPIO = General-purpose input and output (digital)
  - **Input**: program can determine if input signal is a 1 or a 0
  - **Output**: program can set output to 1 or 0
- Can use this to interface with external devices
  - **Input**: switch
  - **Output**: LEDs

Example: STM32 family

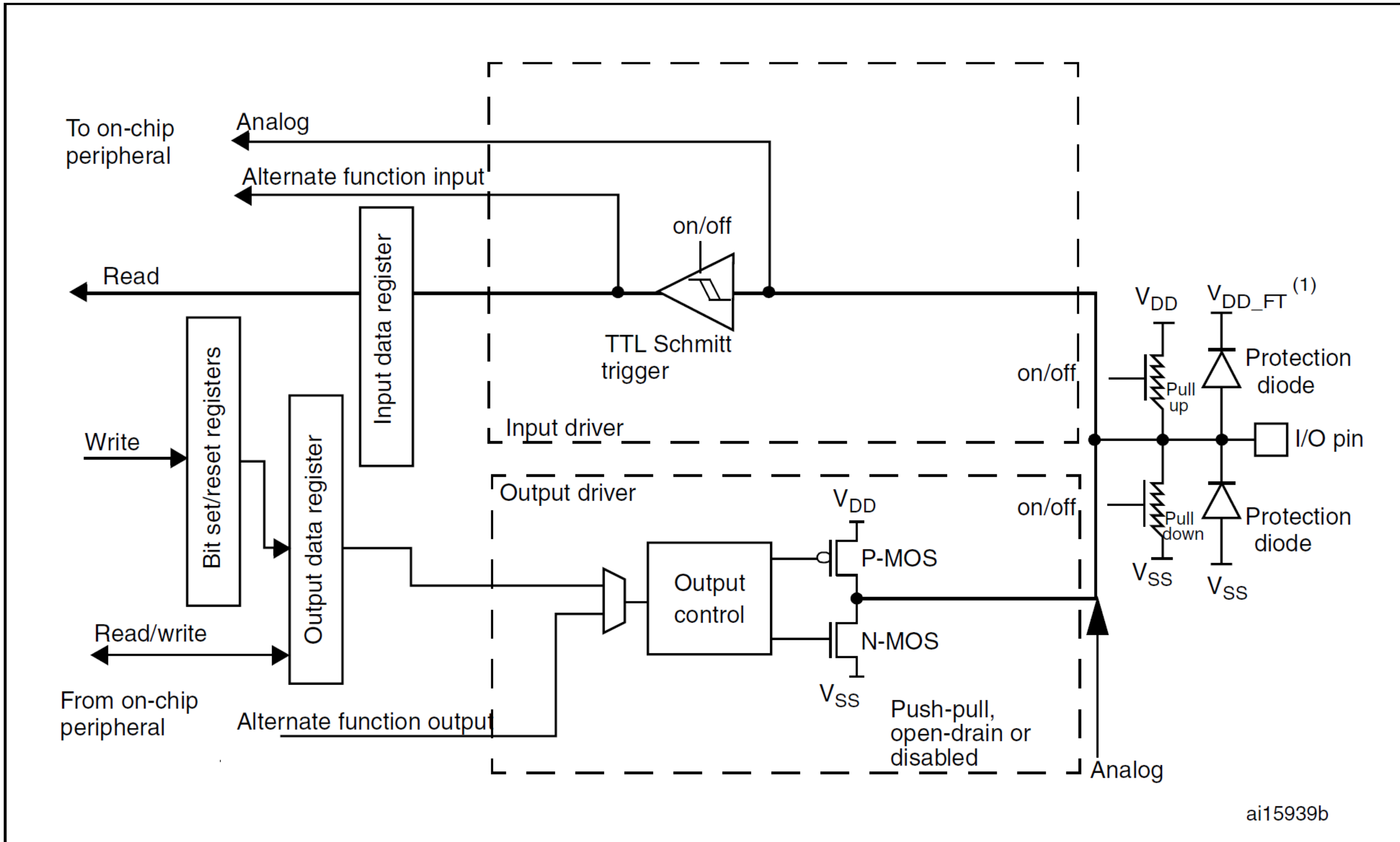
- Microcontrollers of the STM32 family have several **digital ports**, called **GPIO1**, **GPIO2**, **GPIO3**, **...**,
- Each port has **16 bits** and thus **16 electrical pins.** Pins are referred as **Px_y**, where x is the port name (1,2,3, …) and y is the bit (0, 1, ..., 15).
- Example: the pin **P2_3** is the bit 3 of the port 2.
- Each PIN can be configured as **Input** or **Output**
- Some PINs has also an **alternate function**, related to a peripheral e.g. Timer, UART, SPI, etc.

GPIO Alternative Functions

- Pins may have different features
  - To enable an alternate function, set up the appropriate register
- Pins may also have analogue paths for ADC / DAC etc.
- Advantages:
  - Saves space on the package
  - Improves flexibility

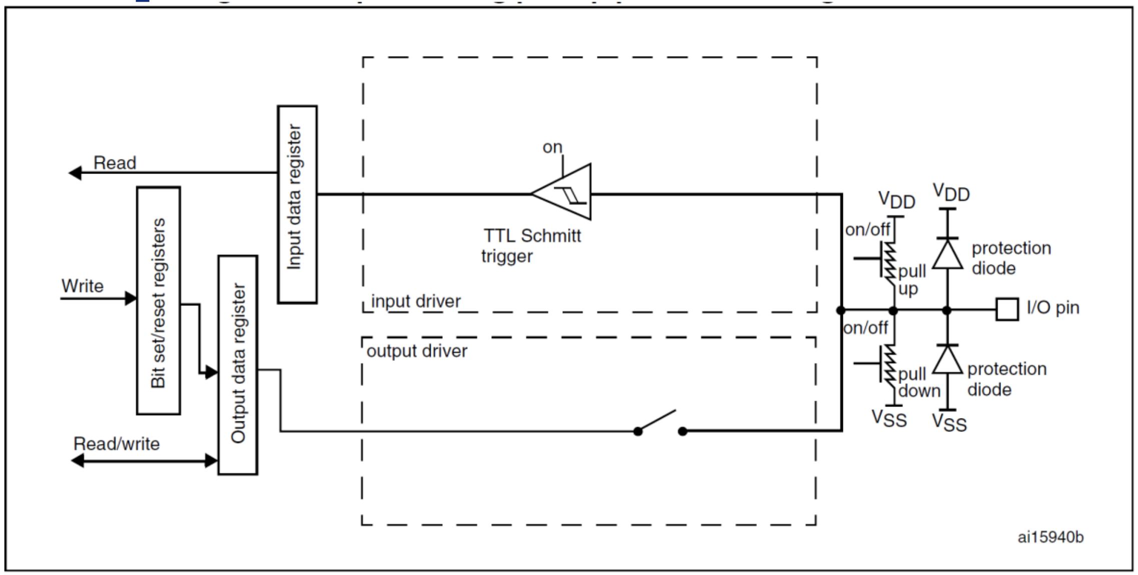


GPIO circuit diagram



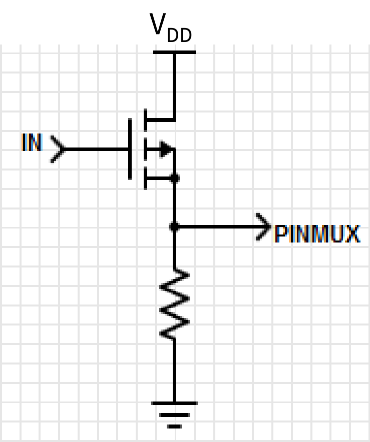
Reference manual - STM32F401xB/C and STM32F401xD/E advanced Arm®-based 32-bit MCUs

GPIO Input Mode

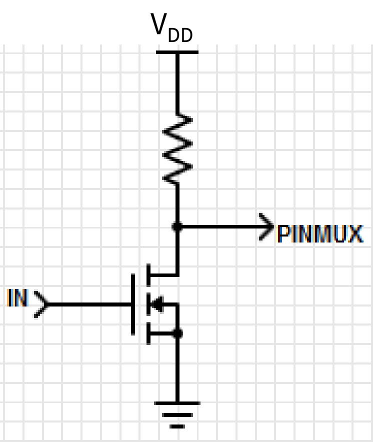


- Ensure a known value on the input of a pin is left floating
- For example, to get the switch SW1 to pull the pin to ground, we should enable the **pull-up**
- In **pull-up mode**, the pin value is:
    - High when SW1 is not pressed
    - Low when SW1 is pressed

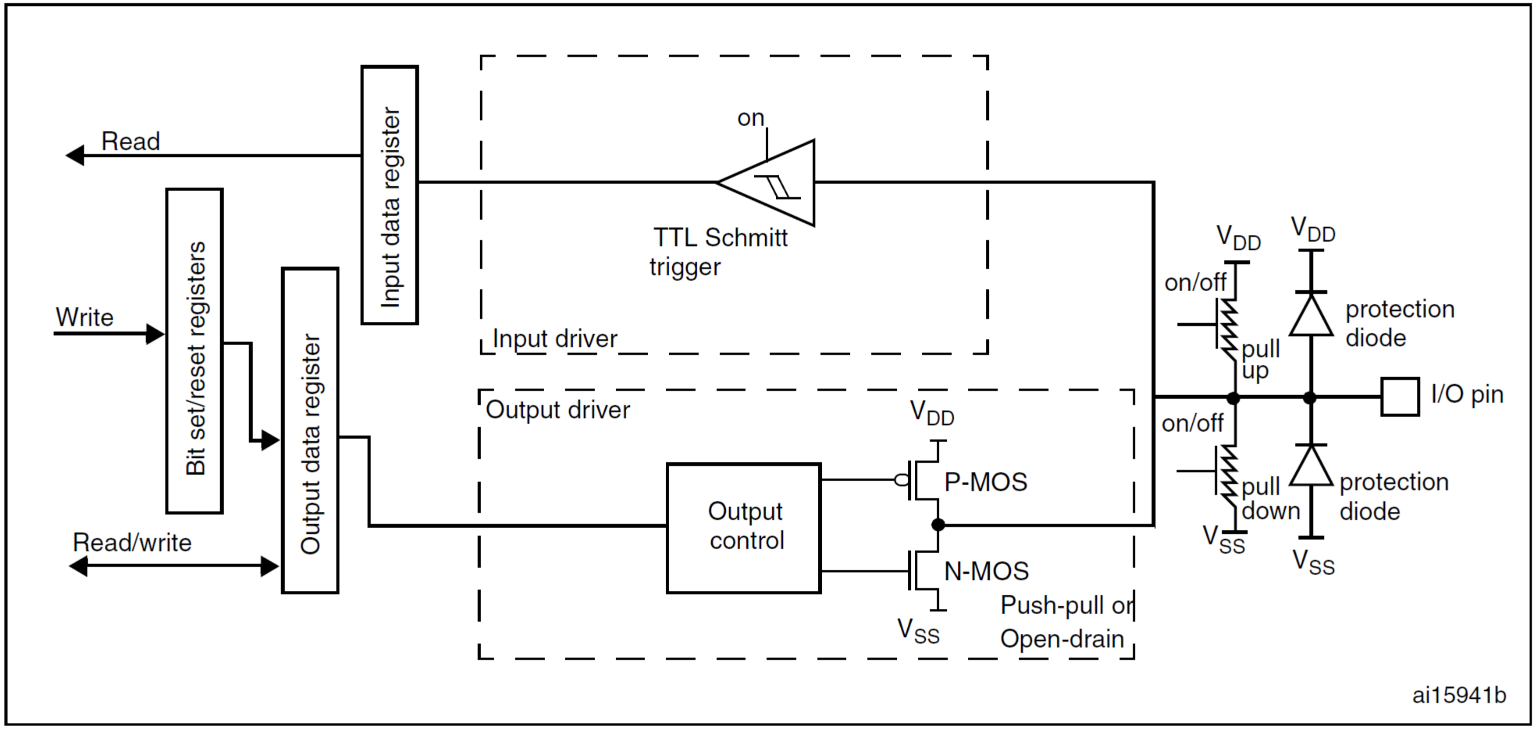
Pull-down



Pull-up

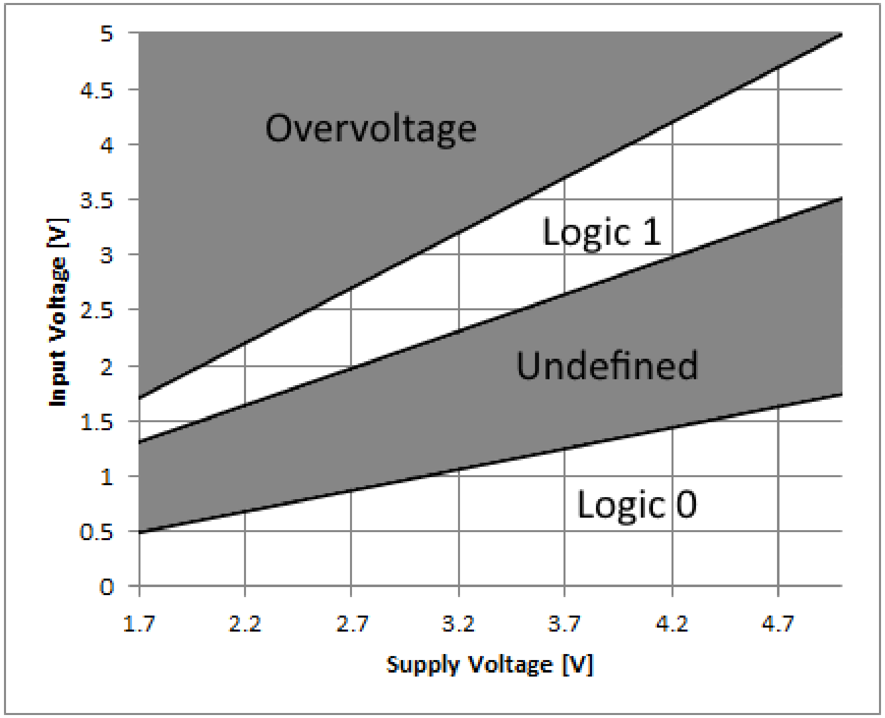


GPIO Output Mode



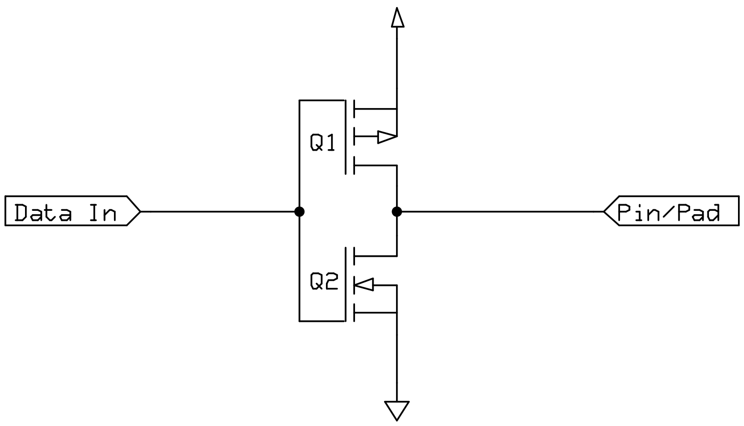
Inputs: What's a One? A Zero?

- Input signal's value is determined by voltage
- Input threshold voltages depend on supply voltage V_{DD}
- Exceeding V_{DD} or GND may damage chip

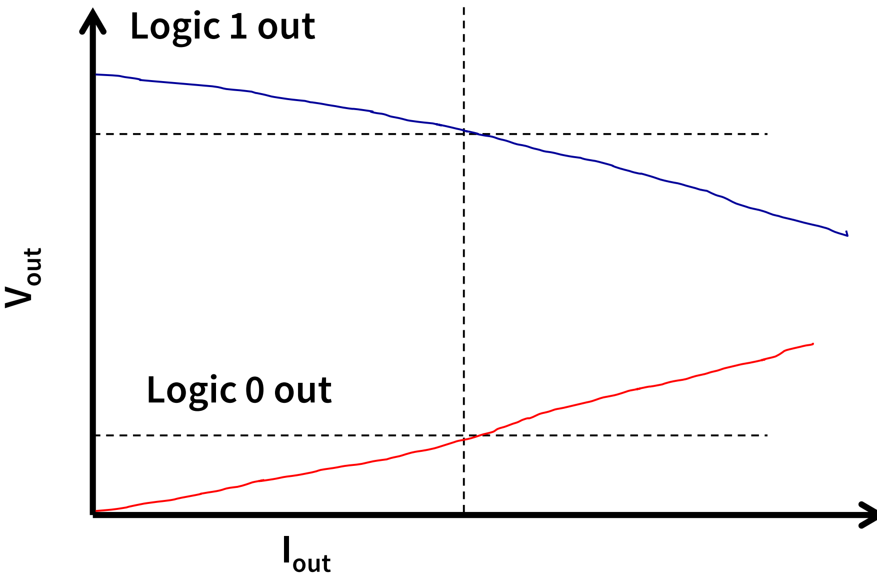


Outputs: What's a One? A Zero?

- Nominal output voltages
  - 1: VDD-0.5 V to VDD
  - 0: 0 to 0.5 V



- Note: Output voltage depends on current drawn by load on pin
  - Need to consider source-to-drain resistance in the transistor
  - Above values only specified when current < 5 mA (18 mA for high-drive pads) and VDD > 2.7 V



Output current

- The ports are not capable of driving loads that require large currents.
    - i.e. the output current is often limited in range of tens of mA.
    - e.g. ARM Cortex-M-based MCUs, the maximum output current per pin is 5 mA.
- If necessary, buffers should be added externally to make sure the current drained from the microcontroller is reasonable.

Controlling the GPIO

3. Controlling the GPIO
   a. Control registers
   b. CMSIS
   c. C-Code

GPIO Special Function Registers

Each general-purpose I/O port has:

- 4 x 32-bit **configuration** registers:
  - GPIOx_MODER: configures each bit as input or output or other
  - GPIOx_OTYPER: output type configuration (push-pull or open-drain)
  - GPIOx_OSPEEDR: configures the maximum frequency of an output pin
  - GPIOx_PUPDR: configures the internal pull-up or pull-down register
- 2 x 32-bit **data** registers:
  - GPIOx_IDR: the input data register
  - GPIOx_ODR: the output data register
- 1 x 32-bit **set/reset** register:
  - GPIOx_BSRR: the bit set/reset register
- 1 x 32-bit **locking** register:
  - GPIOx_LCKR : the bit lock register
- 2 x 32-bit **alternate** function selection register:
  - GPIOx_AFRH
  - GPIOx_AFRL.

Mode Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- MODER allows a programmer to define the functionality of a GPIO pin
- Each pin has 2 bits that permits the following configurations:
  - 00: Input (reset state)
  - 01: General purpose output mode
  - 10: Alternate function mode
  - 11: Analog mode

Pull-up/Pull-down register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **PUPDR** defines the presence of a pull-up or pull-down resistor (or none) at the GPIO pin
- Each pin has **2 bits** that permits the following configurations:
  - **00**: No pull-up/pull-down
  - **01**: Pull-up
  - **10**: Pull-down

IRD/ODR

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
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

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Data Input/Output is performed through the IDR and ODR registers
- Each pin is mapped to the specific bit, so only 16 bits are used in the registers

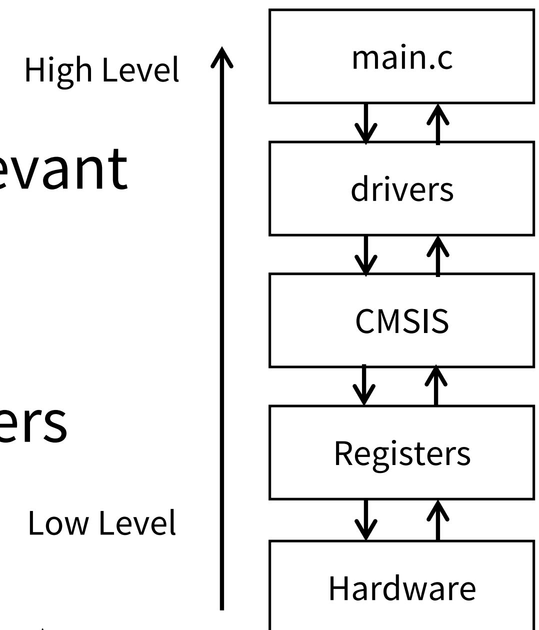
CMSIS to access registers

- It would be tedious to have to look up and remember the addresses of the hardware control registers.
- Instead, we use special **C-language** support.
- The **Cortex Microcontroller Software Interface Standard** (CMSIS) is a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm® Cortex® processors.

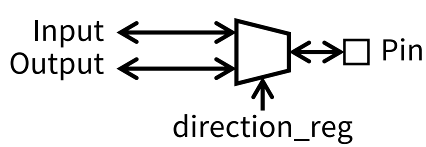
Table 10. STM32F401xD register boundary addresses

Bus	Boundary address	Peripheral
	0xE010 0000 - 0xFFFF FFFF	Reserved
Cortex®-M4	0xE000 0000 - 0xE00F FFFF	Cortex-M4 internal peripherals
	0x5004 0000 - 0xDFFF FFFF	Reserved
AHB2	0x5000 0000 - 0x5003 FFFF	USB OTG FS
AHB1	0x4002 6800 - 0x4FFF FFFF	Reserved
	0x4002 6400 - 0x4002 67FF	DMA2
	0x4002 6000 - 0x4002 63FF	DMA1
	0x4002 5000 - 0x4002 4FFF	Reserved
	0x4002 3C00 - 0x4002 3FFF	Flash interface register
	0x4002 3800 - 0x4002 3BFF	RCC
	0x4002 3400 - 0x4002 37FF	Reserved
	0x4002 3000 - 0x4002 33FF	CRC
	0x4002 2000 - 0x4002 2FFF	Reserved
	0x4002 1C00 - 0x4002 1FFF	GPIOH
	0x4002 1400 - 0x4002 1BFF	Reserved
	0x4002 1000 - 0x4002 13FF	GPIOE
	0x4002 0C00 - 0x4002 0FFF	GPIOD
	0x4002 0800 - 0x4002 0BFF	GPIOC
	0x4002 0400 - 0x4002 07FF	GPIOB
	0x4002 0000 - 0x4002 03FF	GPIOA

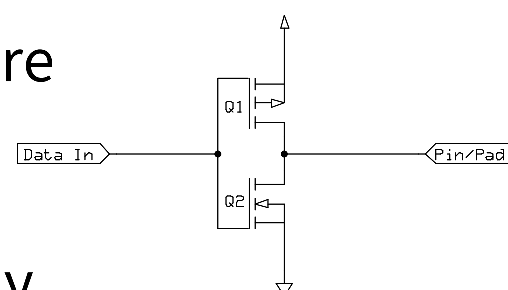
Code Structure



- Main code talks to the drivers, producing easy to read and understand code
  - gpio_set_mode(P2_5, Output)
- Drivers utilise CMSIS library and group relevant actions
  - port_struct->direction_reg = output
- CMSIS transforms memory mapped registers into C structs
  - #define PORT0 ((struct PORT*)0x2000030)
- Registers directly control hardware



- Hardware drives IO pins physically



Drivers Layer: How It Works

```
void gpio_set(Pin pin, int value)
1)    mask = 1 << pin index
2)    tmp = port_struct->data_reg & ~mask
3)    tmp |= value << pin index
4)    port_struct->data_reg = tmp
```

e.g. gpio_set(P2_5, 1) with PORT_DATA_REGISTER = 0b01010101

1. Create a mask for the bit we want to set (0b00100000)
2. Invert the mask (0b11011111) to select all the other bits in the port data register, and save the status of the other bits (tmp = 0b01010101)
3. Move the new value of the bit into position, and or it with the new register value (tmp = 0b01110101)
4. Write the new data register value out to the port (PORT_DATA_REGISTER = 0b01110101)

Drivers Layer: How It Works

int gpio_get(Pin pin)

1) mask = 1 << pin index
2) tmp = port_struct->data_reg & mask
3) tmp >>= pin index
4) return tmp

e.g. gpio_get(P2_5) with PORT_DATA_REGISTER = 0b01110101

1. Create a mask for the bit we want to get (0b00100000)
2. Select the bit in the port data register based on the mask (tmp = 0b00100000)
3. Bitshift the value to produce a one or zero (tmp = 0b00000001)
4. Return the value of the pin back to the user

C Interface: GPIO Configuration

```
/*! This enum describes the directional setup of a GPIO pin. */
typedef enum {
        Reset,   //!< Resets the pin-mode to the default value.
        Input,   //!< Sets the pin as an input with no pull-up or pull-down.
        Output,  //!< Sets the pin as a low impedance output.
        PullUp,  //!< Enables the internal pull-up resistor and sets as input.
        PullDown //!< Enables the internal pull-down resistor and sets as input.
} PinMode;

/*! \brief Configures the output mode of a GPIO pin.
 *  Used to set the GPIO as an input, output, and configure the
 *  possible pull-up or pull-down resistors.
 *  \param pin   Pin to set.
 *  \param mode  New output mode of the pin.*/
void gpio_set_mode(Pin pin, PinMode mode);
```

C Interface: Reading and Writing

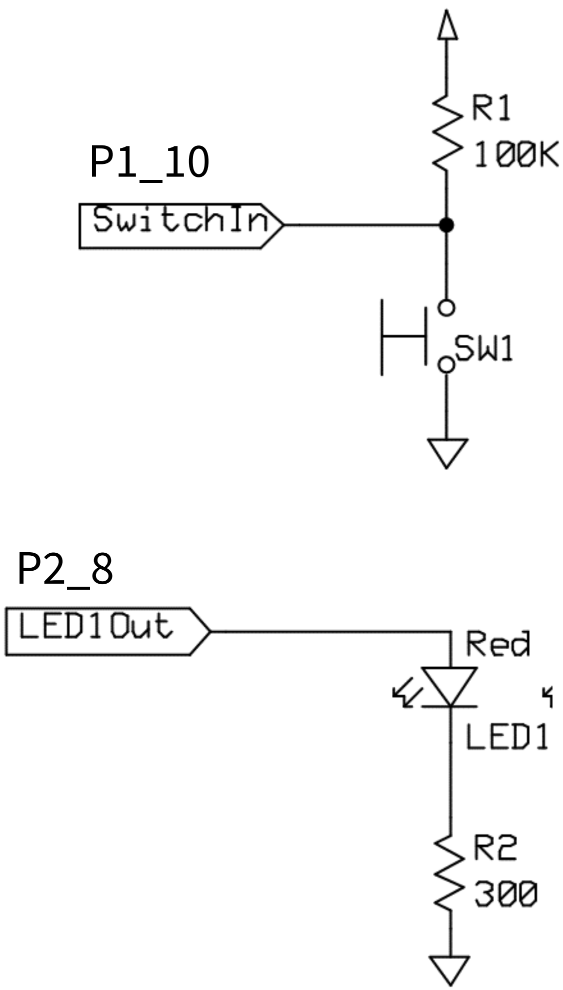
```
/*! \brief Sets a pin to the specified logic level.
 *  \param pin   Pin to set.
 *  \param value New logic level of the pin (0 is low, otherwise high).
 */
void gpio_set(Pin pin, int value);

/*! \brief Get the current logic level of a GPIO pin.
 *  \param  pin   Pin to read.
 *  \return The logic level of the GPIO pin (0 if low, 1 if high).
 */
int gpio_get(Pin pin);
```

Examples

Example 1: Read a pushbutton and lit the LED

```
int main()
{
    // configure pin P1_10 as input
    gpio_set_mode(P1_10, Input);
    // configure pin P2_8 as output
    gpio_set_mode(P2_8, Output);
    // infinite loop
    for (1) {
        int PBstatus=gpio_get(P1_10);
        gpio_set(P2_8, !PBstatus);
    }
}
```



Your turn!

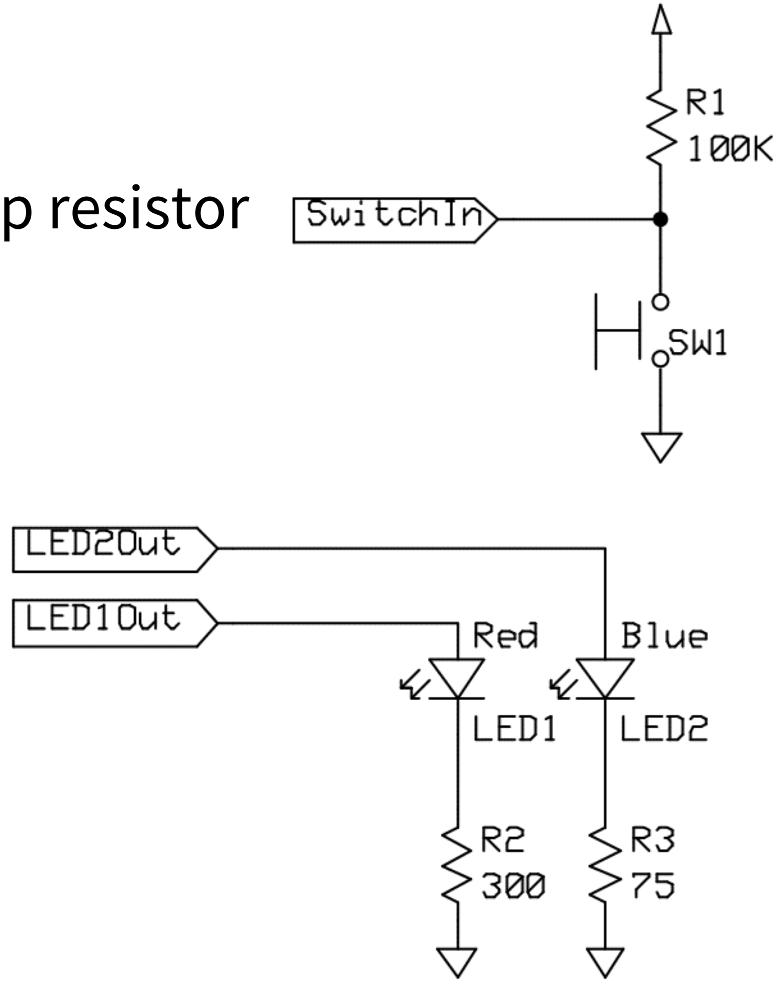
Goal: light only Red LED1 if switch SW1 is pressed and light only Blue LED 2 if it is not

(Write the pseudo code first then the C-code)

Take 10 minutes to write the code before you go to the next slide

Example 2: Pseudo-code

```
Make LED1 and LED2 outputs
Make switch an input with a pull-up resistor
do forever {
        if switch is not pressed {
                Turn off LED1
                Turn on LED2
        } else {
                Turn off LED2
                Turn on LED1
        }
}
```



Example 2: C Code

```
gpio_set_mode(P_LED1, Output); // Set LED pins to outputs
gpio_set_mode(P_LED2, Output);
gpio_set_mode(P_SW, Pullup); // Switch pin to resistive pull-up
while (1) {
	if (gpio_get(P_SW)) {
	// Switch is not pressed (active low), turn LED1 off and LED2 on.
		gpio_set(P_LED1, 0);
		gpio_set(P_LED2, 1);
	} else {
	// Switch is pressed, turn LED2 off and LED1 on.
		gpio_set(P_LED2, 0);
		gpio_set(P_LED1, 1);
	}
}
```