Module 2-A

General Purpose Input/Output (I/O)

Reading

- Textbook, Chapter 14, "General Purpose I/O (GPIO)", pages 341 – 372.
  - We need to talk about this today.

- Textbook, Chapter 11, "Interrupts", pages 237 – 268.
  - We'll talk about this in the next lecture module.

Reading

- STM32F0x1 Family Reference Manual
  - **It's 1008 pages long! Don't read the whole thing.**
  - Read Section 2.2 pages 45 – 50.
    - Be familiar with the memory map.
      - RCC   is at 0x4002 1400. (Section 7.4.15, p142)
      - GPIOA is at 0x4800 0000. (Section 9.4.12, p171)
- Read Section 7.4.6 pages 127 – 128.
  - Be familiar with the RCC register.
- Read Section 9 pages 156 – 172.
  - Be familiar with the GPIO registers.

Learning Outcome #2

"an ability to interface a microcontroller to various devices."

How?

A. General Purpose I/O
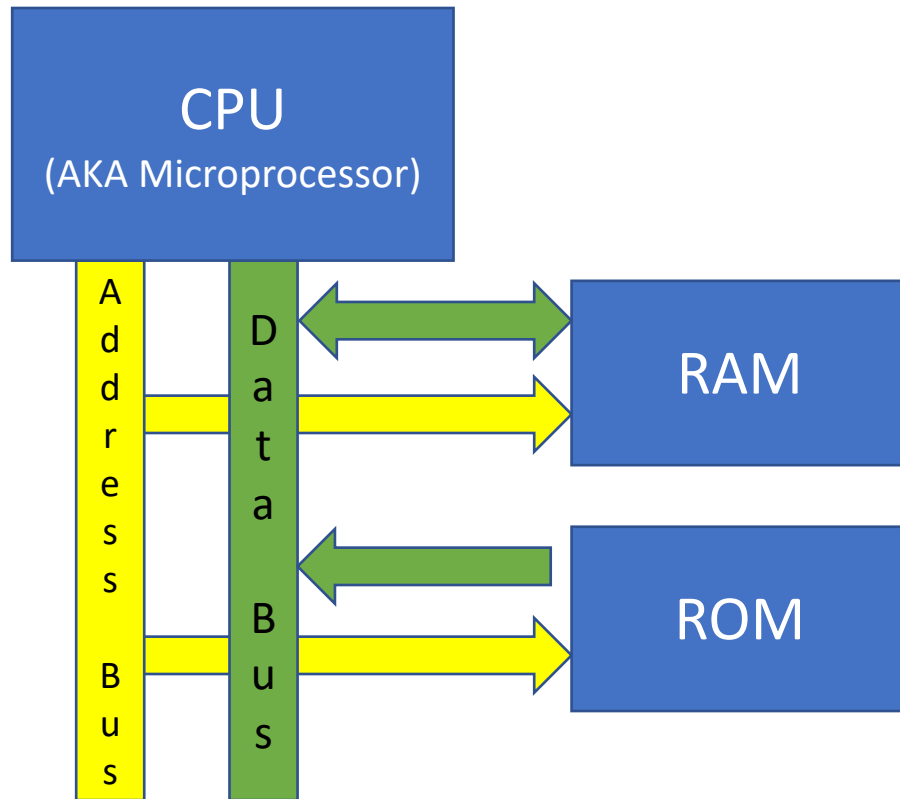B. Interrupts and Exceptions
C. Basic Timers
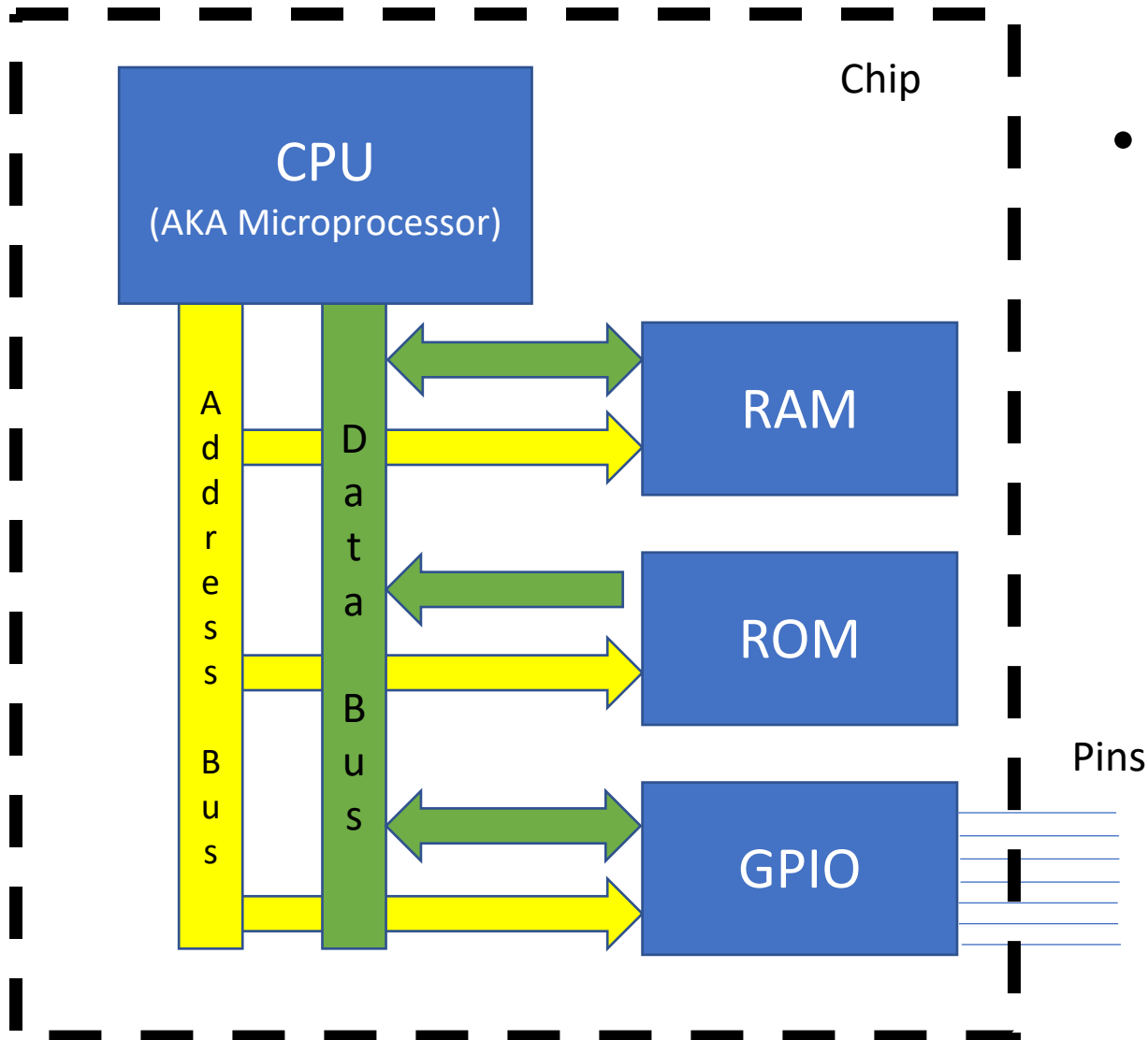D. Embedded C
E. Advanced Timers
F. Debouncing and Multiplexing
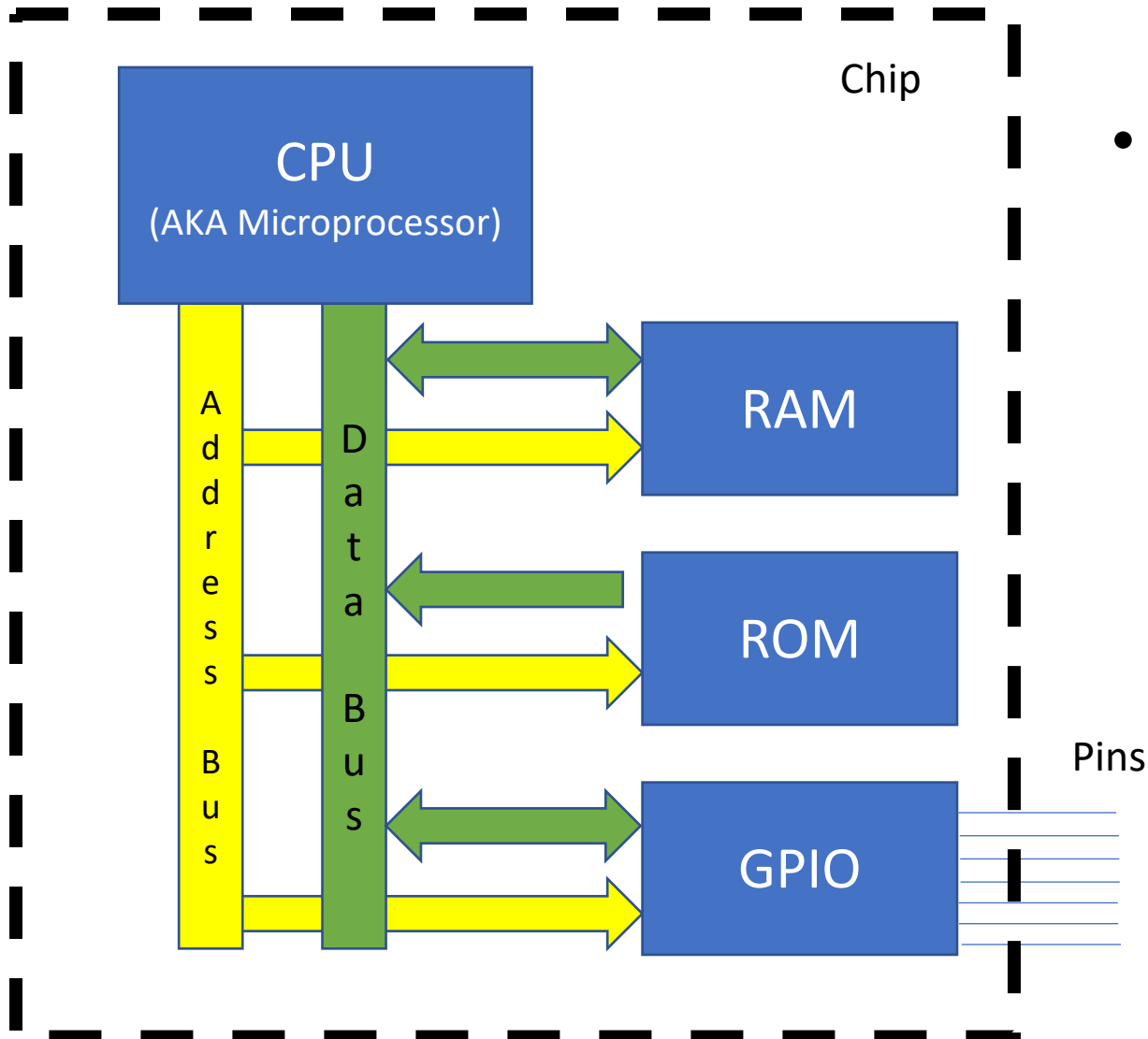
General Microprocessor Interfacing

- CPU needs to talk to RAM (read/write variables)
- Read from ROM (read-only data - like the program)
- One bus to drive the address
- One bus to send/receive data

General Microprocessor Interfacing

Chip

CPU
(AKA Microprocessor)

Address Bus

Data Bus

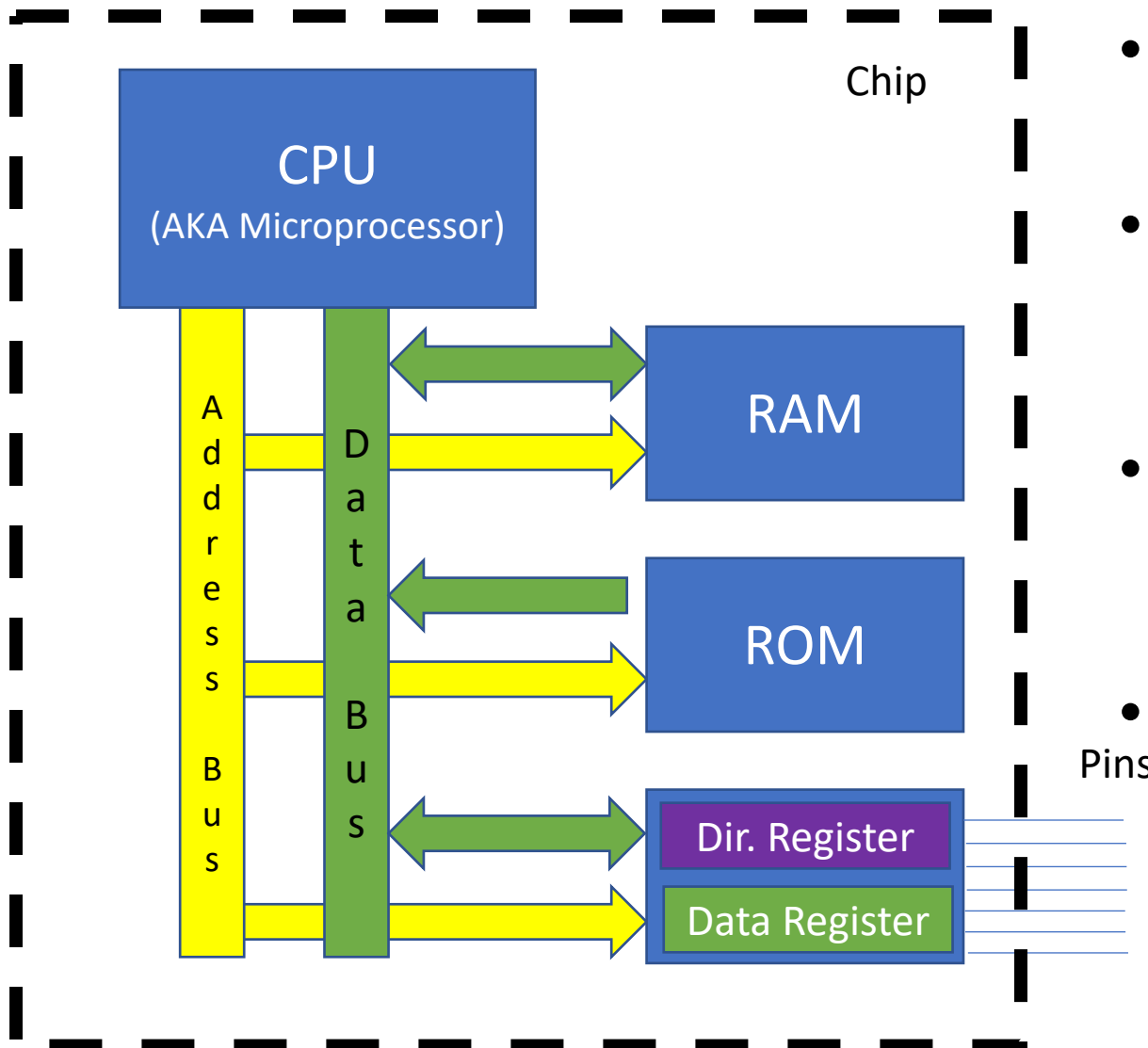RAM

ROM

GPIO

Pins

- Want to talk to other stuff too!
  - Drive 1/0 on pins
  - Read 1/0 from pins

General Microprocessor Interfacing

- How do I drive/read these pins?
  - Bi-directional
  - CPU will write stuff to them (like to blink and LED)
  - CPU will read stuff from them (like a button push).

General Microprocessor Interfacing


- There is an on-chip module outside the CPU that contains "registers"
- One drives the direction of the pins
  - i.e. if there are 8 pins, 8 bits where 0/1 says in/out for each pin
- The other stores the data
  - An 8-bit register with 1-bit for each pin's value.
- CPU accesses these registers through LDR/STR instructions.
  - Certain memory addresses correspond to each register.

Our chip: The STM32F091

- CPU
  - This is the only thing we have been looking at so far
- Flash ROM
- SRAM
- A whole wack of peripherals!
- Today we are looking at two systems:
  - GPIO – what we really want
  - Reset and Clock Control (RCC) – something we have to setup to get what we want
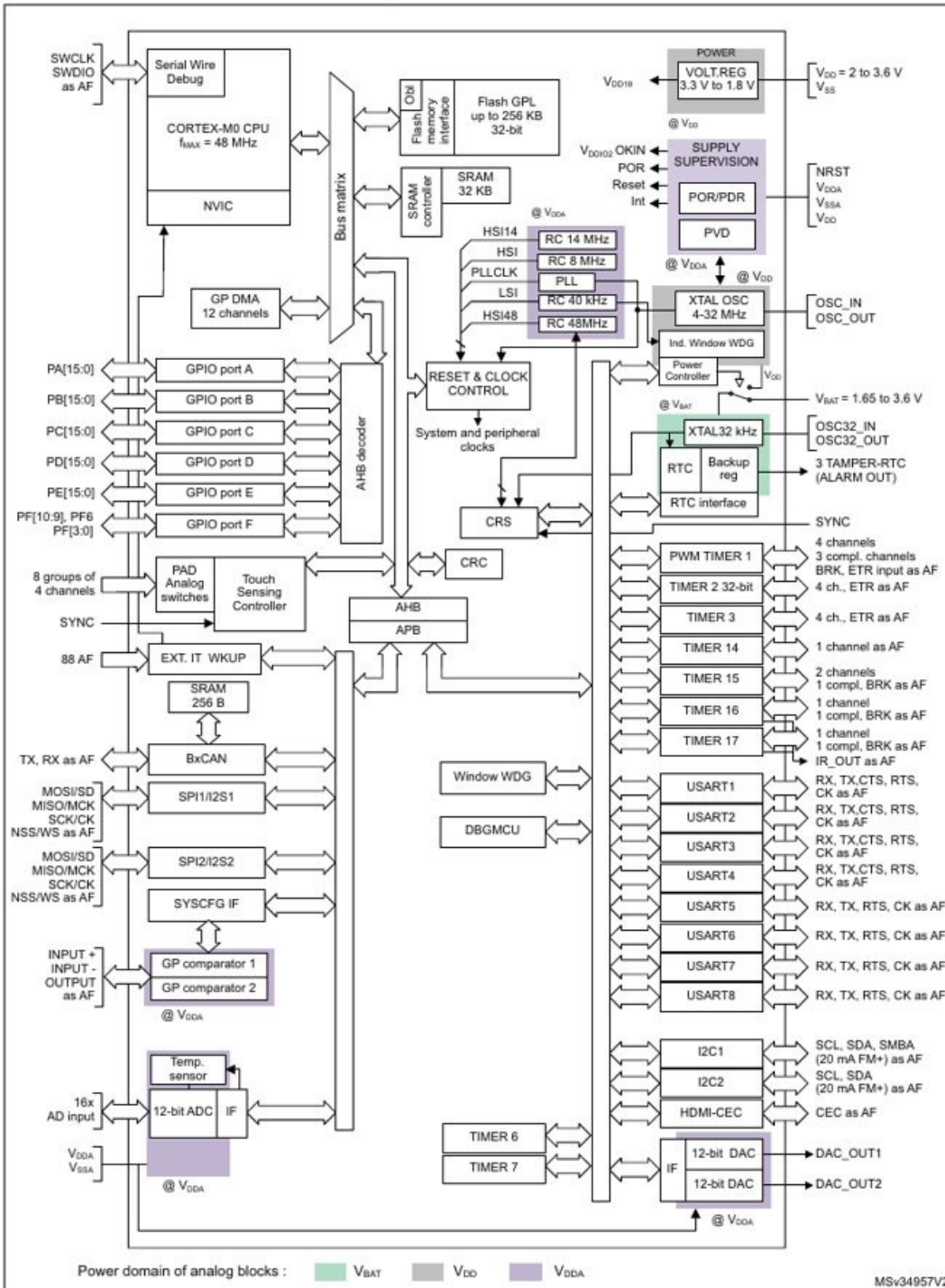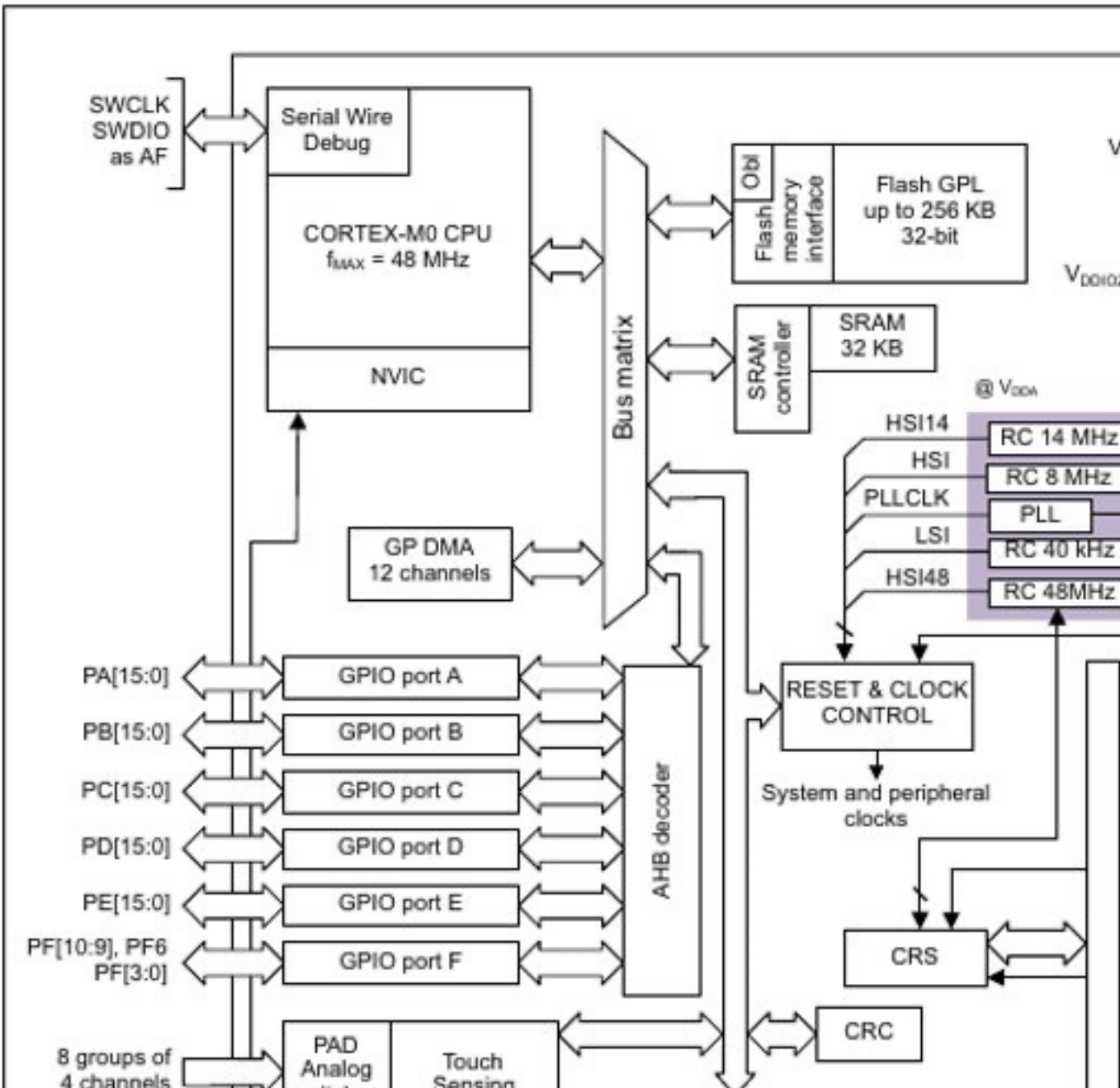
GPIO subsystem

- Multiple "ports", most of which are 16-bits wide.
  - Primarily, A, B, and C.
  - Each port has multiple control registers.
  - By default, ports are disabled to save power.
  - Enable them with the Reset and Clock Control.
    - RCC is also a block of registers mapped into memory. See STM32F0 Family Reference, Section 2.2, Page 45.

RCC: Reset & Clock Control

- RCC is a collection of 32-bit control registers.
  - See FRM, Table 1, page 46 for address ranges. Or...
  - See STM32F091xBC datasheet, Table 20, page 46 for address ranges.
  - **Remember: Not the same thing as CPU registers. We access the register with loads/stores to memory!**
  - The RCC registers are mapped in addresses 0x4002 1000 – 0x4002 1038.
    - See STM32F0 Family Reference, Section 7.4.15, page 142 for the list of registers.
  - Today, we care only about RCC_AHBENR, the AHB Enable Register.
    - Offset **0x14** from base address of RCC (0x4002 1000).
    - See STM32F0 Family Reference, Section 7.4.6, page 127, for long description.

0x14	RCC_AHBENR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	IOPFEN	IOPEEN	USART4RST	IOPCEN	IOPBEN	IOPAEN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRCEN	Res.	FLITFEN	Res.	SRAMEN	DAM2EN	DMAEN
	Reset value								0		0	0	0	0	0	0										0		1		1	0	0

7.4.6 AHB peripheral clock enable register (RCC_AHBENR)

Address offset: 0x14

Reset value: 0x0000 0014

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

Note: *When the peripheral clock is not active, the peripheral register values may not be readable by software and the returned value is always 0x0.*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.
							rw		rw	rw	rw	rw	rw	rw	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRC EN	Res.	FLITF EN	Res.	SRAM EN	DMA2 EN	DMA EN
									rw		rw		rw	rw	rw

Bits 31:25 Reserved, must be kept at reset value.

Bit 24 **TSCEN:** Touch sensing controller clock enable
Set and cleared by software.
   0: TSC clock disabled
   1: TSC clock enabled

Bit 23 Reserved, must be kept at reset value.

Bit 22 **IOPFEN:** I/O port F clock enable
Set and cleared by software.
   0: I/O port F clock disabled
   1: I/O port F clock enabled

Bit 21 **IOPEEN:** I/O port E clock enable
Set and cleared by software.
   0: I/O port E clock disabled
   1: I/O port E clock enabled

Bit 20 **IOPDEN:** I/O port D clock enable
Set and cleared by software.
   0: I/O port D clock disabled
   1: I/O port D clock enabled

Bit 19 **IOPCEN:** I/O port C clock enable
Set and cleared by software.
   0: I/O port C clock disabled
   1: I/O port C clock enabled

Bit 18 **IOPBEN:** I/O port B clock enable
Set and cleared by software.
   0: I/O port B clock disabled
   1: I/O port B clock enabled

AHBENR description

- Documentation for each 32-bit memory location:
  - Offset
  - Initial value
  - Access methods
  - Name/position of each bit
    - "rw" means readable and writable
    - ("rw" is explained in FRM 1.1 on page 41)
  - Long description of each bit

Let's turn on a port

- To use a port, we must enable the clock connected to it first.
    - E.g. to enable Port C, we want to set IOPCEN, bit 19.
    - **Caution**: We can't just write the value 0x080000 (1<<19) into the register, because it would turn off the bits of other important things.
        - SRAM EN keeps SRAM enabled in sleep mode. We sort of need that.

0x14	RCC_AHBENR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	IOPFEN	IOPEEN	USART4RST	IOPCEN	IOPBEN	IOPAEN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRCEN	Res.	FLITFEN	Res.	SRAMEN	DAM2EN	DMAEN
	Reset value								0		0	0	0	0	0	0										0		1		1	0	0

Only set the bits we care

1. Read in the current value of the control register (with a LDR instruction) .

   - Value gets stored in a temporary CPU register

2. Logical **OR** *the* current value with the bits we want on.

3. Store it back to the control register (with a STR instruction)

Example code enabling Port C

```
1  .equ RCC,      0x40021000  // Base address of RCC control registers
2  .equ AHBENR, 0x14          // Offset of the RCC_AHBENR
3  .equ IOPCEN, 0x80000       // Value to enable "IO Port C"
4
5  ldr   r0, =RCC             // Load, into R0, base address of RCC
6  ldr   r1, [r0, #AHBENR]    // Load, into R1, value of RCC offset by AHBENR
7  ldr   r2, =IOPCEN          // Load, into R2, value to enable Port C
8  orrs  r1, r2               // Combine R2 into R1
9  str   r1, [r0, #AHBENR]    // Store new bits into RCC_AHBENR
```

LDR, OR, STR:
We will use this repeatedly to
set control register bits

Are we done? Can we do I/O now?

- Nope
  - All we did was enable the clock.
  - Still need to setup the port
- Each GPIO port is configured with a group of 32-bit control registers in a contiguous range of memory.
  - See datasheet Table 20, (or FRM Table 1) page 46 for the address ranges:
    - Port A registers are in the addresses 0x4800 0000 – 0x4800 002C.
    - Port B registers are in the addresses 0x4800 0400 – 0x4800 042C.
    - Port C registers are in the addresses 0x4800 0800 – 0x4800 082C.
  - For the control register descriptions, see FRM Section 9.4.12, on page 171.
    - Offset 0x00: MODER: Mode: Input or output? (Note: special reset value for Port A)
    - Offset 0x0C: PUPDR: Pull-up/Pull-Down?
    - Offset 0x10: IDR: Input Data Register (16 bits)
    - Offset 0x14: ODR: Output Data Register (16 bits)

PORT MODER

- 16 2-bit values determine, input, output, or special operation.
    - 00: Port pin is used for input
    - 01: Port pin is used for output
    - 10: Port pin has an alternate function
    - 11: Port pin is an analog pin
- What is the default configuration of MODER for Port C?

0x00	GPIOx_MODER (where x = B..F)	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[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]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pins input by default: Need to set them to output

- We can configure the port for output by setting the 2-bit entries to '01'.
  - **Caution**: Pins 13, 14, and 15 of port C are a little special. Keep those configured as inputs.
    - We'll use these when we connect an external 32.768 kHz crystal oscillator. These specific pins are easy to damage if they are used as outputs. After that, you cannot use an external crystal.
    - It gives us a justification to talk about ORing and ANDing values.
- 'OR' the values in as we did with the RCC.

Configuring Port C as an Output

```
1  .equ GPIOC,  0x48000800 // Base address of GPIOC control registers
2  .equ MODER,  0x00        // Offset of the GPIOC_MODER
3  .equ OUT0_12,0x01555555 // Value to enable Port C bits 0 - 12
4
5  ldr  r0, =GPIOC          // Load, into R0, base address of GPIOC
6  ldr  r1, [r0, #MODER]    // Load, into R1, value of GPIOC offset by MODER
7  ldr  r2, =OUT0_12        // Load, into R2, value to enable PC0 … PC12
8  orrs r1, r2              // Combine R2 into R1
9  str  r1, [r0, #MODER]    // Store new bits into GPIOC_MODER
```

Is this enough? Can I output stuff now?

- Yes!
- The Output Data Register (ODR) is a 32-bit register. The lower 16 bits matter.
- Anything you write to the lower 16 bits of the ODR will be transmitted to the pins we have configured, in the MODER, to be outputs.
  - Anything written to the upper 16 bits of the ODR will have no effect.
  - Anything written to ODR bits configured as inputs will have no effect. You can write a '1' to a bit of the ODR that is configured as an input with MODER. It won't turn into an output.
  - We configured port C so pins 0 – 12 were outputs. If we write 0xff00 to the ODR,
    - pins PC0 – PC7 would be pulled low
    - pins PC8 – PC12 would be pushed high
    - pins PC13 – PC15 would not be driven high or low because they are inputs
  - We can read the ODR to find out the last value we wrote to it.

GPIO Circuitry
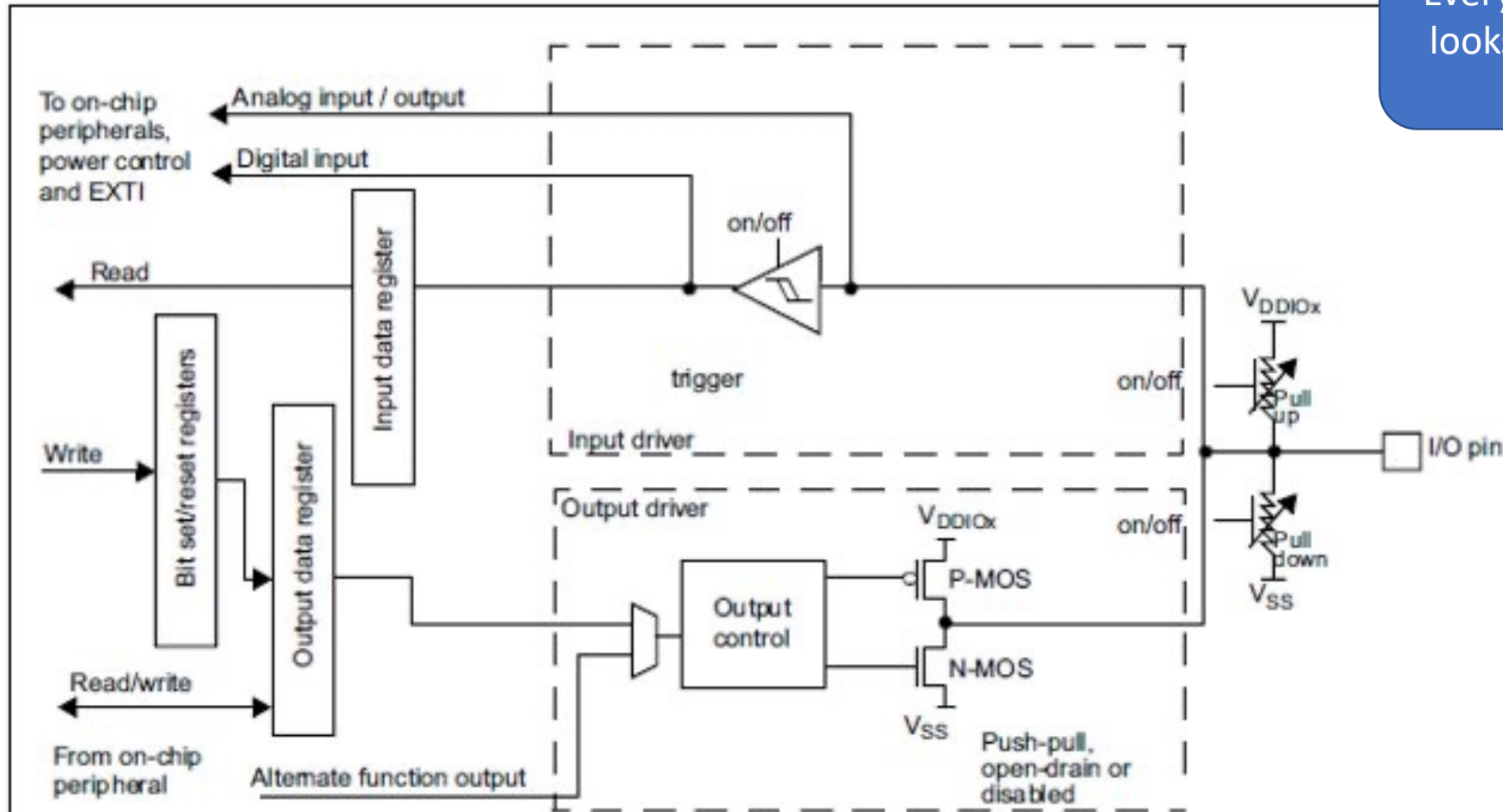
Every GPIO Pin looks like this.

STR instruction sets bit in the ODR

Controller drives MOSFETS – drive 1 or 0

Example turning bits on and off

```
 1   // After configuring RCC AHBENR and PORTC MODER, then we can...
 2   .equ GPIOC, 0x48000800  // Base address of GPIOC control registers
 3   .equ ODR, 0x14          // Offset of the GPIOC_MODER
 4   .equ PC8, 0x100         // Port C pin 8
 5   .equ PC89, 0x300        // Port C pins 8 and 9
 6
 7       ldr   r3, =GPIOC     // Load, into R3, base address of GPIOC
 8       ldr   r2, =PC8       // The bit to turn on PC8
 9       ldr   r1, =PC89      // Bits to toggle PC8 and PC9
10   again:
11       str   r2, [r3, #ODR] // Store bit pattern into ODR
12       ldr   r0, =1000000000// Load one billion
13       bl    nano_wait      // Wait one billion nanoseconds
14       ldr   r2, [r3, #ODR] // Get the last value written to ODR
15       eors  r2, r1         // XOR it to toggle PC8 and PC9
16       b     again          // Do it again
17   nano_wait:
18       subs  r0,#83
19       bgt   nano_wait
20       bx    lr
```

Let's do input!

- We set the Port's MODER register to '00' for each pin that we want to use as an input.
  - We can still write 1s and 0s to the ODR for those bits, and the pin will not drive those values.

Table 25. GPIO register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOA_MODER	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[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]	
	Reset value	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Circuit for Input
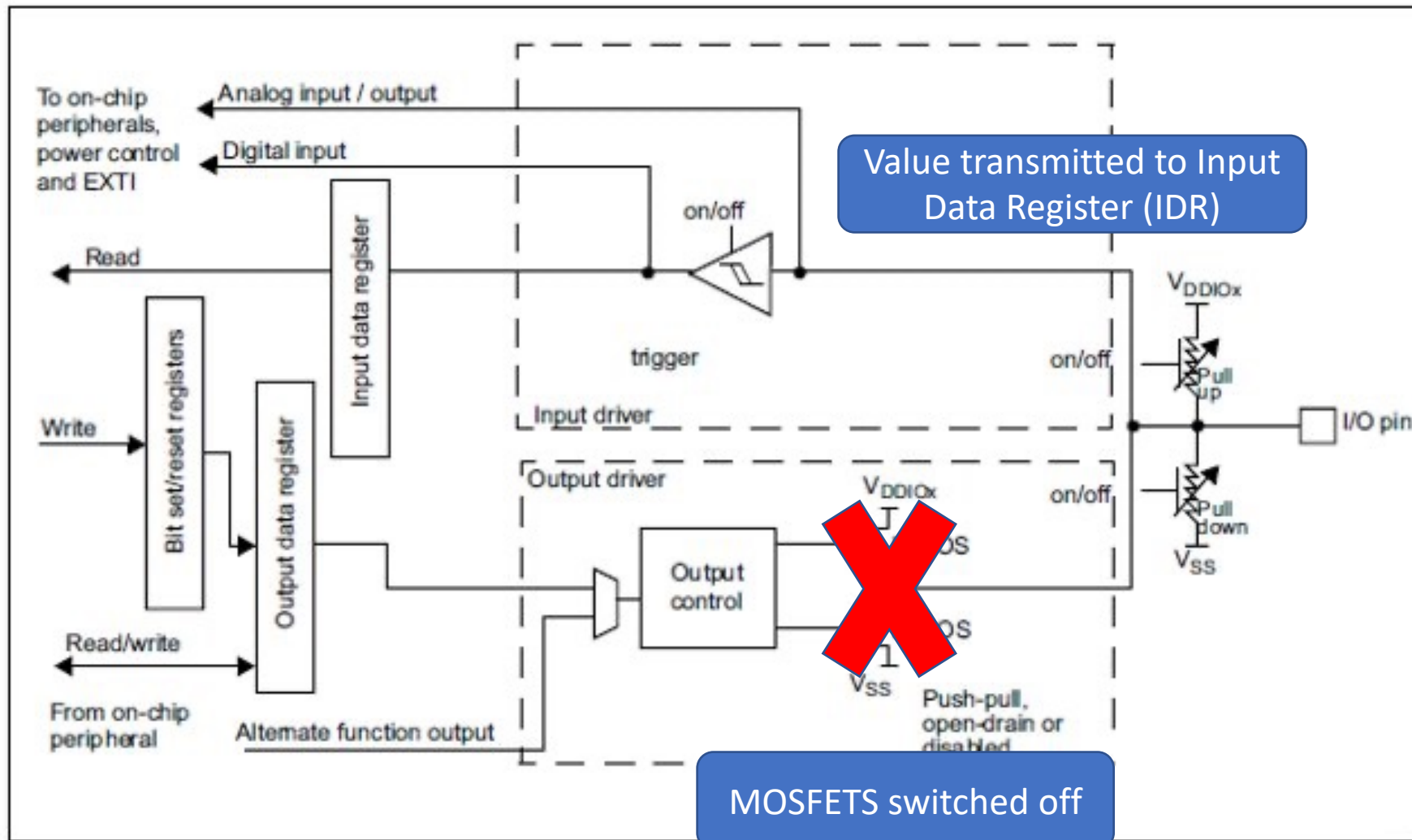
Configure Port A as input

- **Assume, first, that we ORed 0x20000 into RCC_AHBENR!**
- We can clear all the 2-bit fields for every pin we want to act as an input.
  - **Caution**: There are some pins of port A we need to leave alone.
    - PA13 and PA14 are programming pins. If you turn those both into inputs, the debugger will stop working.
      - You will do this by accident, and you will be confused.
  - Need to selectively clear bits in the MODER in a manner similar to how we selectively set bits in MODER for writing.

Example on setting Port A as input

```
1  .equ GPIOA,   0x48000000 // Base address of GPIOA control registers
2  .equ MODER,   0x00        // Offset of the GPIOA_MODER
3  .equ CLR0_12,0x03ffffff   // Value to clear from MODER
4
5  ldr   r0, =GPIOA          // Load, into R0, base address of GPIOA
6  ldr   r1, [r0, #MODER]    // Load, into R1, value of GPIOA offset by MODER
7  ldr   r2, =CLR0_12        // Load, into R2, value to clear
8  bics  r1, r2              // Clear the bits in R1 that are set in R2
9  str   r1, [r0, #MODER]    // Store new bits into GPIOA_MODER
```

Even though the default is input – always set these just in case.
Some test cases mess with defaults

Example to copy PA0 (push button) to PC8 (green LED)

```
1   .equ GPIOA, 0x48000000   // Base address of GPIOA control registers
2   .equ GPIOC, 0x48000800   // Base address of GPIOC control registers
3   .equ IDR, 0x10           // Offset of the IDR
4   .equ ODR, 0x14           // Offset of the ODR
5
6       ldr   r0, =GPIOA        // Load, into R0, base address of GPIOA
7       ldr   r1, =GPIOC        // Load, into R1, base address of GPIOC
8       movs r3, #1
9   again:
10      ldr   r2, [r0, #IDR]  // Load, into R2, value of GPIOA_IDR
11      ands r2, r3            // Look at only bit zero.
12      lsls r2, #8            // Translate bit 0 to bit 8
13      str   r2, [r1, #ODR]  // Store value to GPIOC_ODR
14      b     again
```
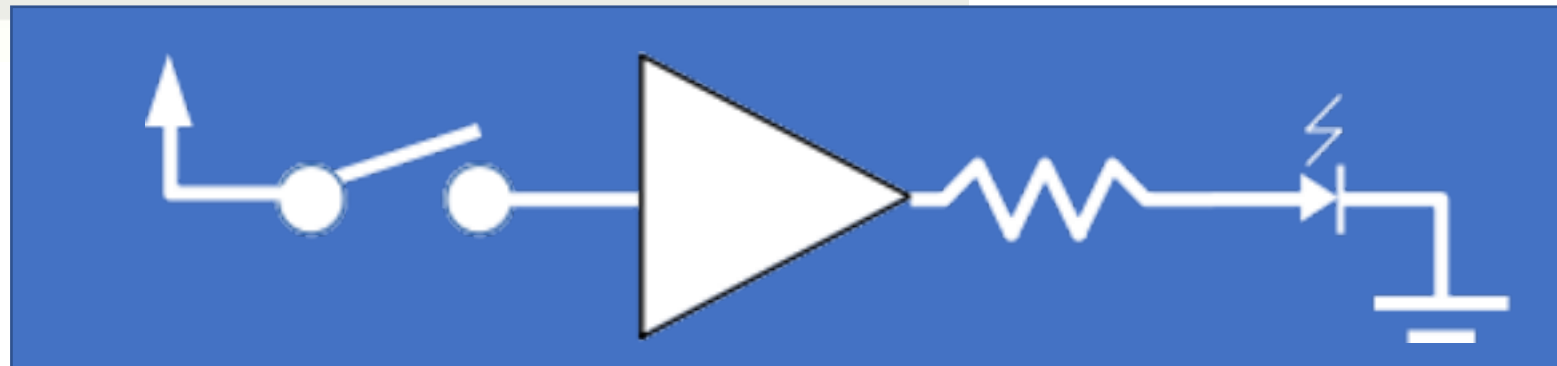
Electrical Problem: Input will stay high

- Note that the switch connected to the input of the buffer can only bring the input high.

- Assuming that the buffer input is very high resistance (it is), then once it is charged up it will stay charged.

- Opening the switch never turns the input off.

- We need another resistor to drain the charge off the input.
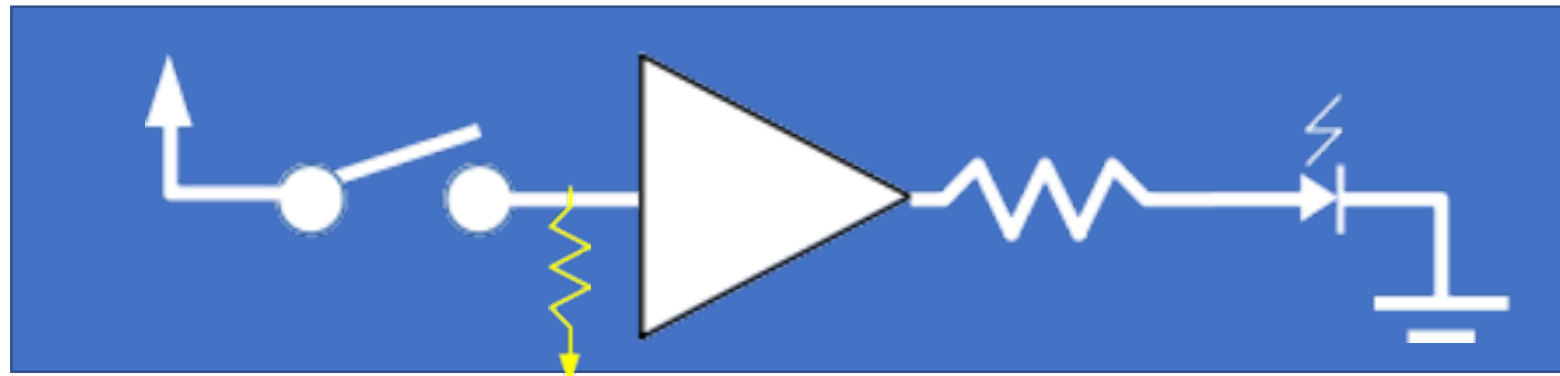
Circuit is designed to deal with this
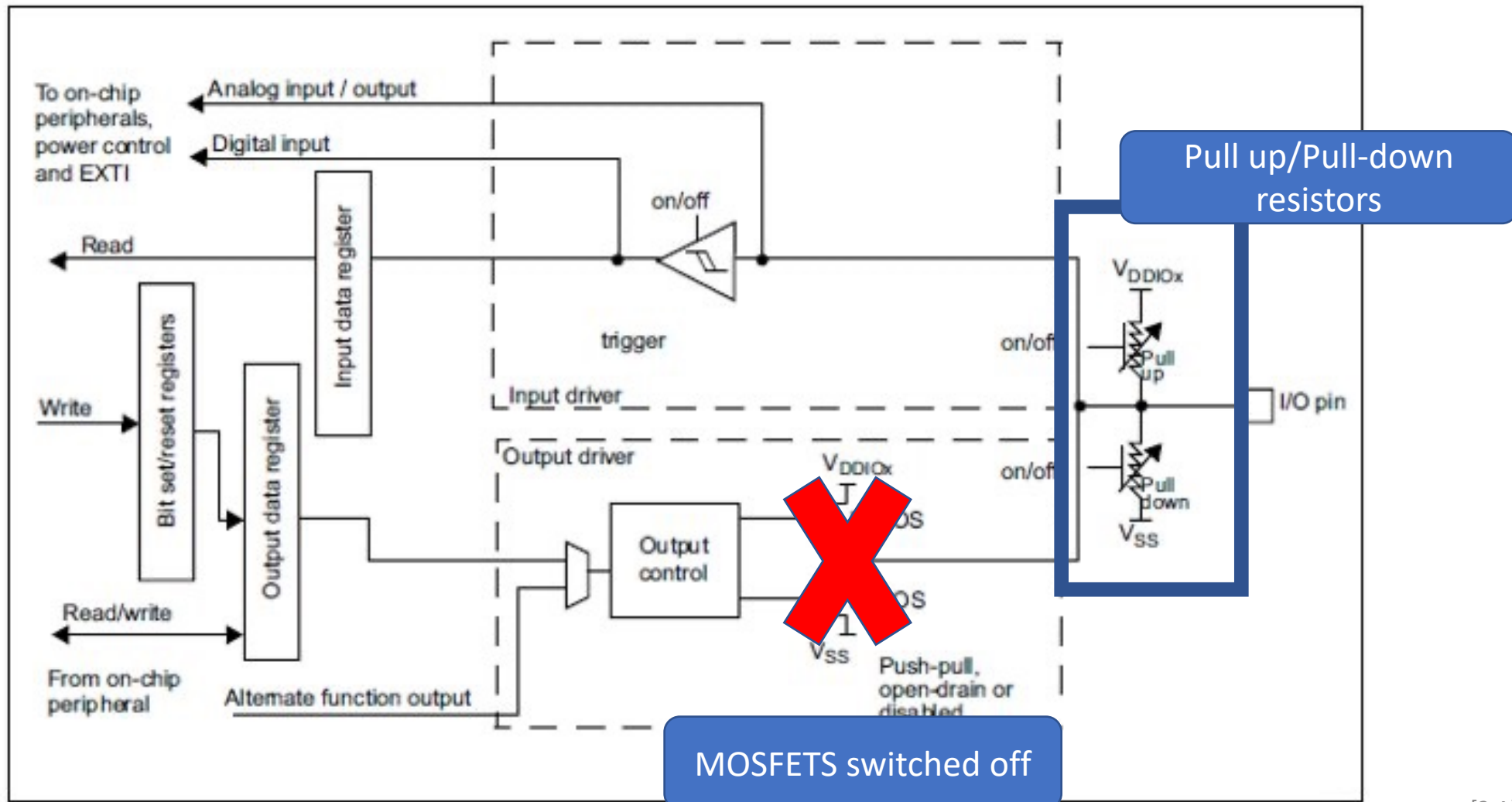
Pull-up/Pull-down resistors

- Gently pull the input high or low in cases where it is not being driven. AKA floating. E.g.:
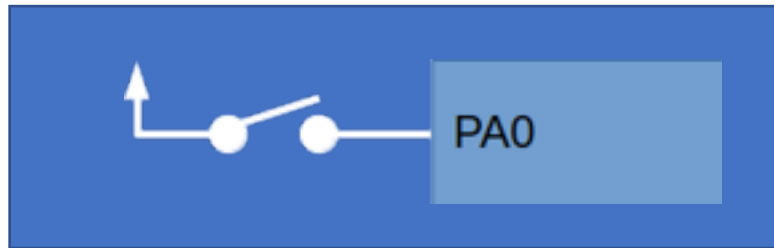
- Usually, we want PA0 to read 'low' when the button is not pressed. By default, it floats. That probably means it will stay high forever.
  - (BTW: It was deliberately designed like that to make you use a pull-down.)
- You must configure the PUPDR to pull it low.

GPIOx_PUPDR

- Offset 0x0C from GPIOx
- Two bits per pin (like GPIOx_MODER)
  - 00: No pull-up or pull-down
  - 01: Pull-up
  - 10: Pull-down
  - 11: "Reserved"

Steps to read PA0 (Switch 2)

- First, turn on the IOPAEN bit in the RCC AHBENR
  - Without this, PORTA registers will do nothing.
- Clear the two bits in PORTA MODER that correspond to PA0
  - This turns PA0 into an input.
- Set the bits in the PORTA PUPDR for PA0 to enable the internal pull-down resistor
  - This makes the IDR read zero when the button is not pressed.
- Read the PORTA IDR and check bit 0 to find the state of PA0.
  - Repeat as often as needed.

Remember this?
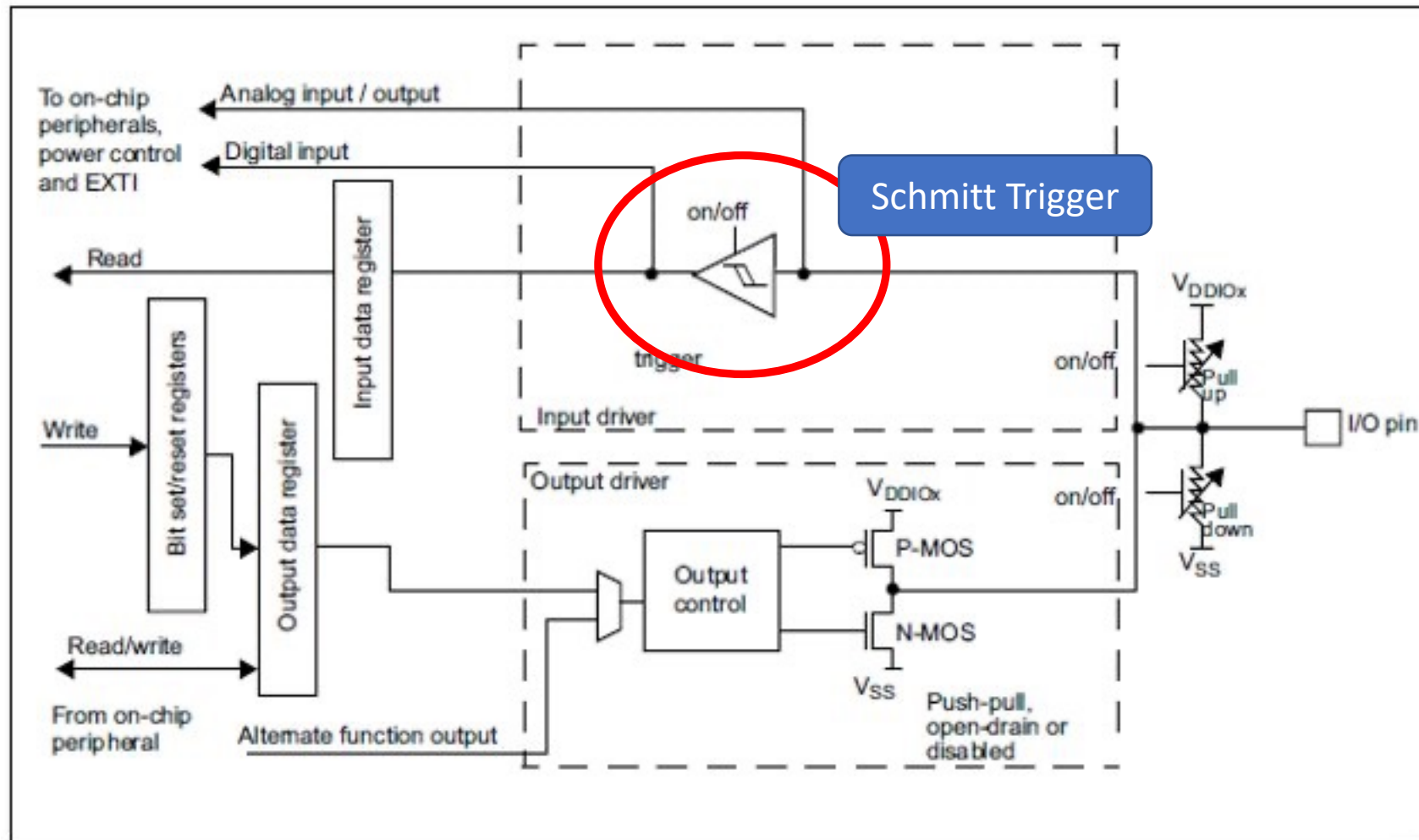
Schmitt Trigger

- A non-inverting buffer that is immune from problems with metastability for slowly-changing input values.
  - You know that logic gates have a maximum speed at which they can respond.
  - There is also a minimum rise/fall time (also known as "slew rate") for inputs to ensure that metastability does not result.
  - Too slow means the signal spends too much time in the logic "gray zone".
  - Input reading can oscillate between high and low.

Schmitt Trigger Example
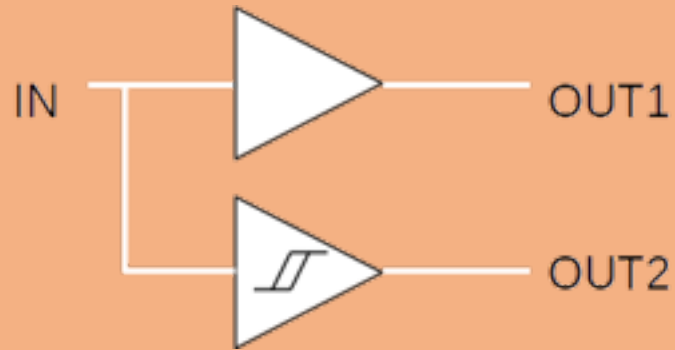
Ideally, buffer should not register a '1' or a '0' until it is above the minimum high or below the minimum low.
Regular gates may flip if the input stays in the middle too long.
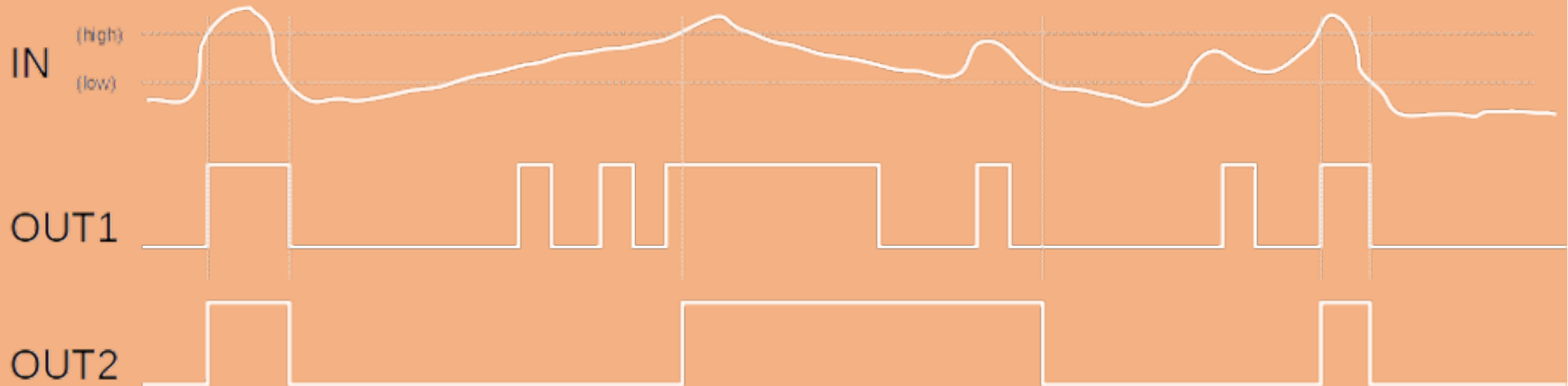
More about Inputs

- Input is sampled into the IDR on every AHB clock.
- Input is still sampled even when a pin is configured to be an output.
  - Why would the IDR differ from the ODR?

Other GPIO registers: OTYPER

- OTYPER: configure a pin for push-pull or open-drain driving.
  - Only 16 bits. One bit per pin:
    - 0: Push-pull
    - 1: Open-Drain
  - In Open-Drain configuration, the P-channel MOSFET is inactive.
    - 0 in ODR: pull low
    - 1 in ODR: Don't drive it
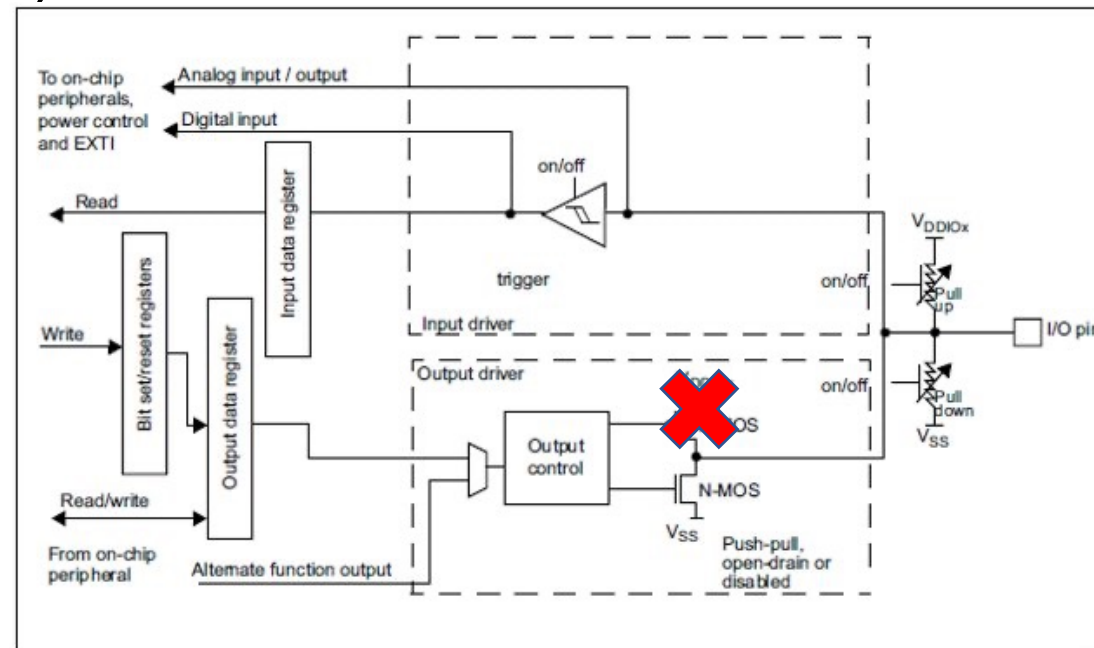
Using an Open-Drain pin
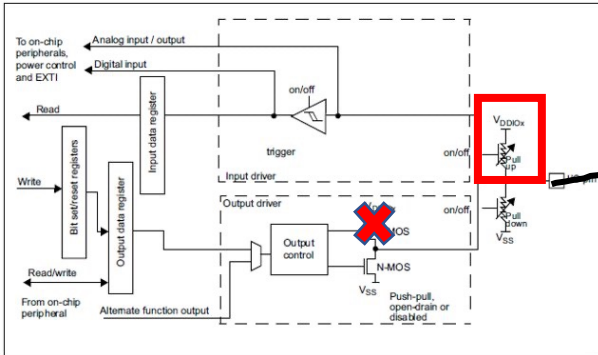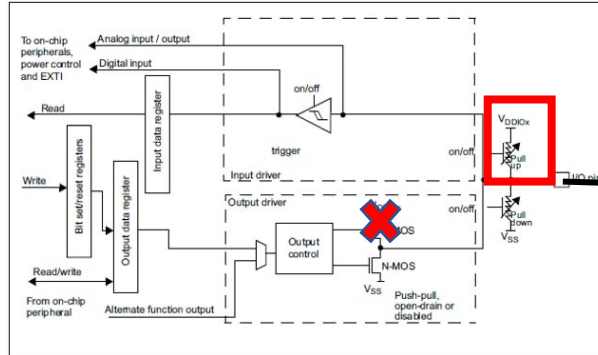
- **One-wire communication.**
  - All MCUs use a pull-up resistor.
  - All MCUs configure pins for Open-Drain.
    - Any one can pull the wire low by writing to the ODR.
  - All MCUs can see the state of the wire by reading the IDR.
  - This is how I^{2}C works.

Other GPIO Registers: OSPEEDR

For some of the devices we drive - we will have to set it to high speed.
Downside of high-speed: More Power

- OSPEEDR: Configure the speed (or slew rate) of output.
  - 32-bit register. 2-bits per pin.
    - x0: Low speed (default)
    - 01: Medium speed
    - 11: High speed
  - What does this mean?
    - In STM32 Family Reference Manual Section 9.4.3 (page 166) it says "Refer to device datasheet…"
    - Anybody look at the device data sheet?
      - Table 55. I/O AC characteristics, Page 83.
    - When driving low-current loads at low speed, it won't matter. When driving high current DC loads (LEDs, or many digital inputs) or when changing the pin at high rate (>2 MHz) it will affect the slope of the rise and fall time.

Other GPIO registers: BSRR/BRR

- BRR: (Bit Reset Register)
  - For each '1' bit written to the lower 16 bits of BRR, turn OFF the corresponding ODR bit.

- BSRR: (Bit Set & Reset Register)
  - For each '1' bit written to the lower 16 bits of BSRR, turn ON the corresponding ODR bit.
  - And for each '1' bit written to the higher 16 bits of BSRR, turn OFF the corresponding ODR bit.

BRR/BSRR detail

Writes to BRR/BSRR

- With a single write to BRR, you can turn off selected bits in the ODR. (No read, mask, AND, write back.)

- With a single write to BSRR, you **can turn on and turn off** selected bits in the ODR.

    - An **ATOMIC** operation!

Atomic means "indivisible"

- There are situations where you cannot reliably use multiple instructions to do something.
  - Operating system mechanisms
    - Locking & Mutual exclusion
  - CPUs doing I/O with interrupts enabled.
    - We'll discuss interrupts next lecture.

- Want a way to do things with a single instruction.

Examples of BRR / BSRR

```
1  / Assume current ODR is
2  //        (0xF0F0)    1111000011110000
3  ldr r0, =GPIOC
4  ldr r1, =0x3113 // 0011000100010011
5  str r1, [r0, #BRR]
6
7
8  // New value of ODR is
9  //                    1100000011100000
```

```
1  // Assume current ODR is
2  //                (0xF0F0) 1111000011110000
3  ldr r0, =GPIOC
4  ldr r1, =0x30100302 (-0011000000010000)
5                       (+0000001100000010)
6  str r1, [r0, #BSRR]
7
8  // New value of ODR is
9  //                (0xC3E2) 1100001111100010
```

- BRR / BSRR make it easy to do complicated ODR updates fast.

What GPIO can do

- Well, it's general, so pretty much anything, given enough pins
  - Bus protocols can be implemented using it
    - RAM/ROM controller.
  - LCD controller
  - Simple serial connections
  - SPI Protocol: SDCard Interface
  - Implementing stuff with GPIO: Sometimes called "bit banging"

Do Not Use The Following Pins

- External oscillator pins:
  - PC13, PC14, PC15: 32.768 kHz crystal (configured through J4)
  - PF0, PF1: 8MHz crystal (configured through J5)
- PA13, PA14: Programming Pins
- PF11: System Boot Pin (use only as an output, not as an input)
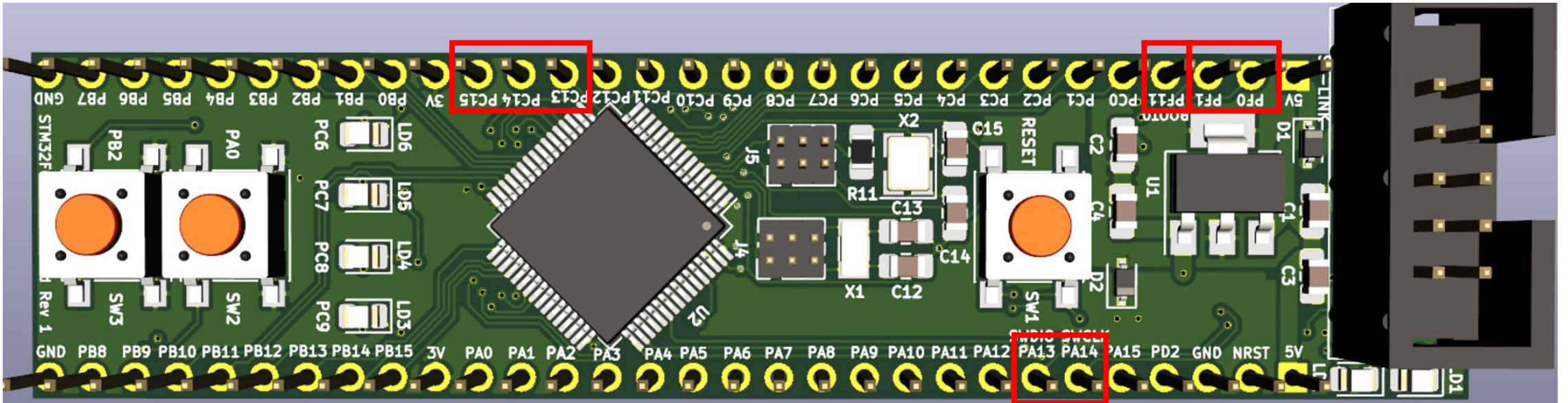
Already Connected For You

- NRST: Pushbutton SW1
- PA0: User Pushbutton SW2
- PB2: User Pushbutton SW3

- PC6: User LED LD6 (Red LED)
- PC7: User LED LD7 (Yellow LED)
- PC8: User LED LD8 (Green LED)
- PC9: User LED LD9 (Blue LED)
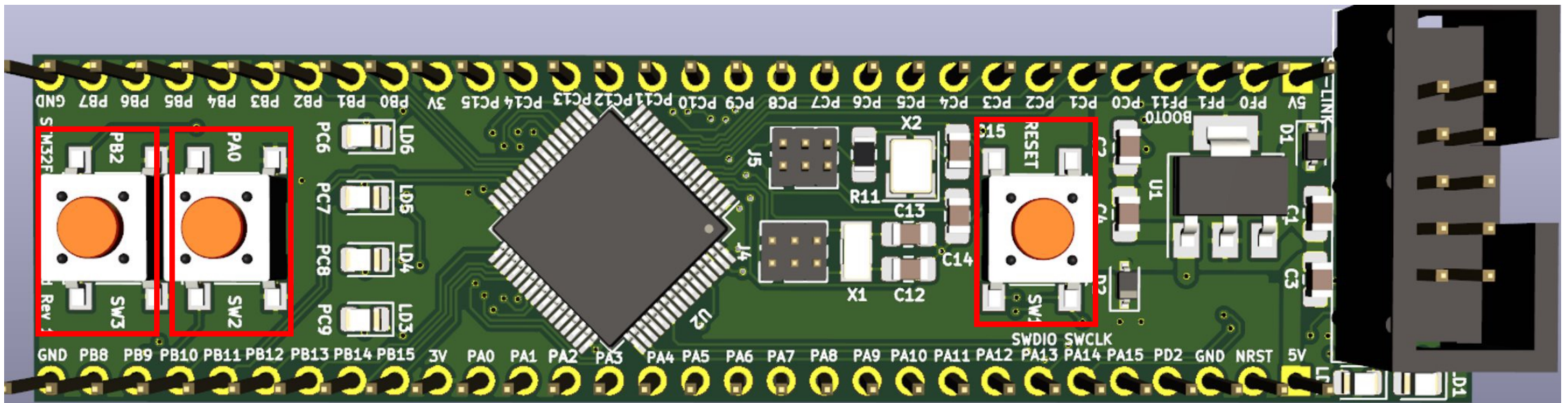
How to Wreck Your Dev Board

- (It's not as bad as our previous development board.)
- Be careful with D1 and D2. You'll break them off.
- When pushing the dev board into a breadboard, press it down using buttons SW1, SW2, SW3.
- Never connect a power source >3.6V to the "3V" pins or any I/O pins.