General Purpose I/O

ECE 362
https://engineering.purdue.edu/ece362/

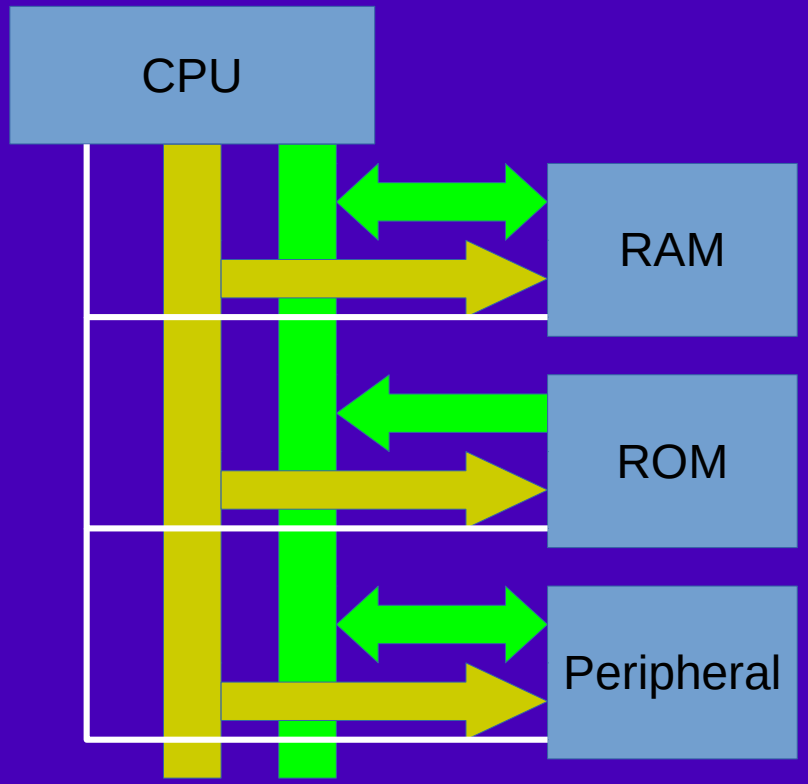
Reading Assignment

- 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 Assignment

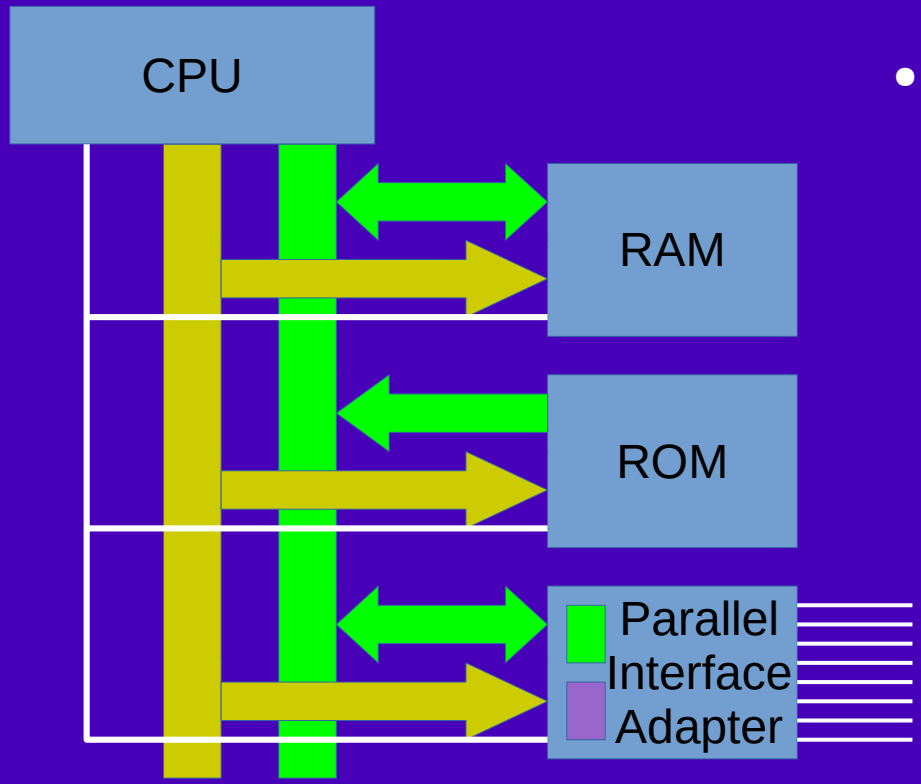
- STM32F0x1 Family Reference Manual
  - It's 1008 pages long! Don't read the whole thing.
  - Read Section 2.2 pages 45 – 50.
    - Be familiar with the memory map.
      - RCC is at `0x4002 1400`. (Section 7.4.15, p142)
      - GPIOA is at `0x4800 0000`. (Section 9.4.12, p171)
  - Read Section 7.4.6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

Microprocessor Interfacing



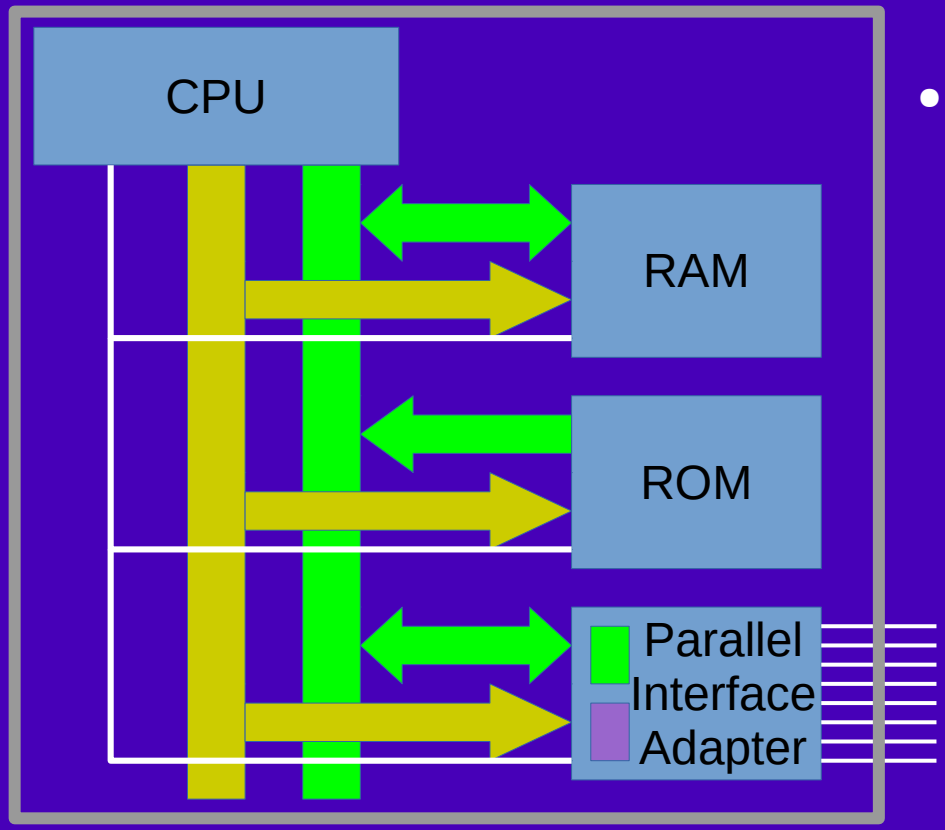
- Classic microprocessor interfaces were simple:
  - (Bidirectional) Data Bus
  - Address Bus
  - Control Lines.
- Peripherals that acted like memory. "Memory Mapped"

Microprocessor Interfacing



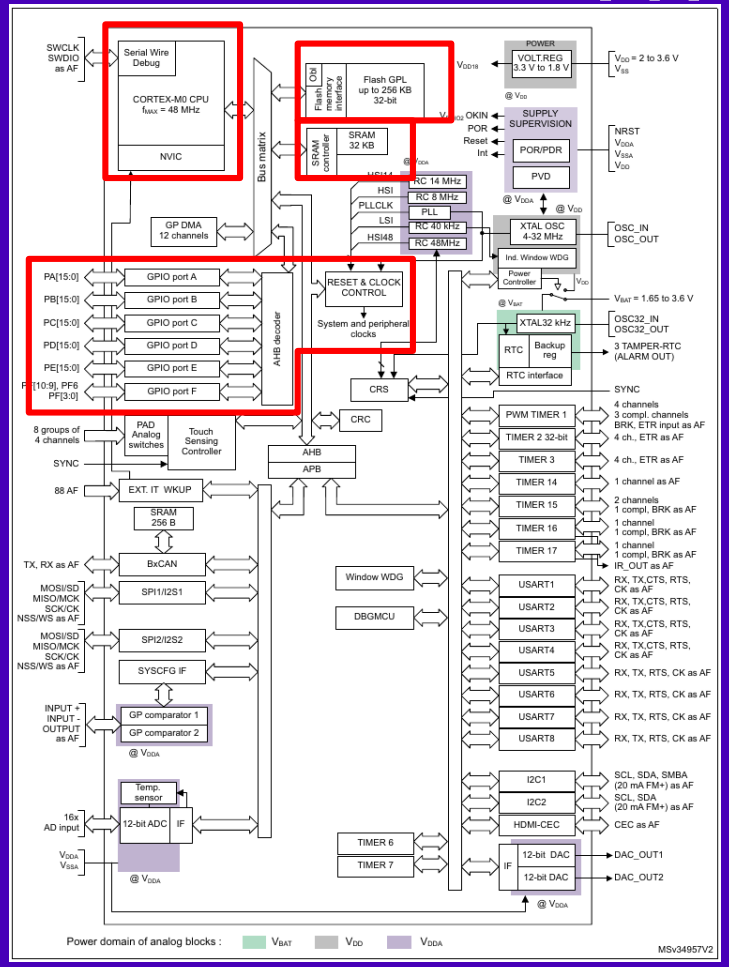
- Then people started building standardized peripherals to do parallel data I/O.
  - Eight bits at a time.
  - One register to decide the direction of the lines.
  - One register to read and write the lines.

Parallel Interface Adapter



- This was so useful that they started building the whole thing on a single chip.
  - And it was called a microcontroller.
  - We have the same things today, but with more complexity.

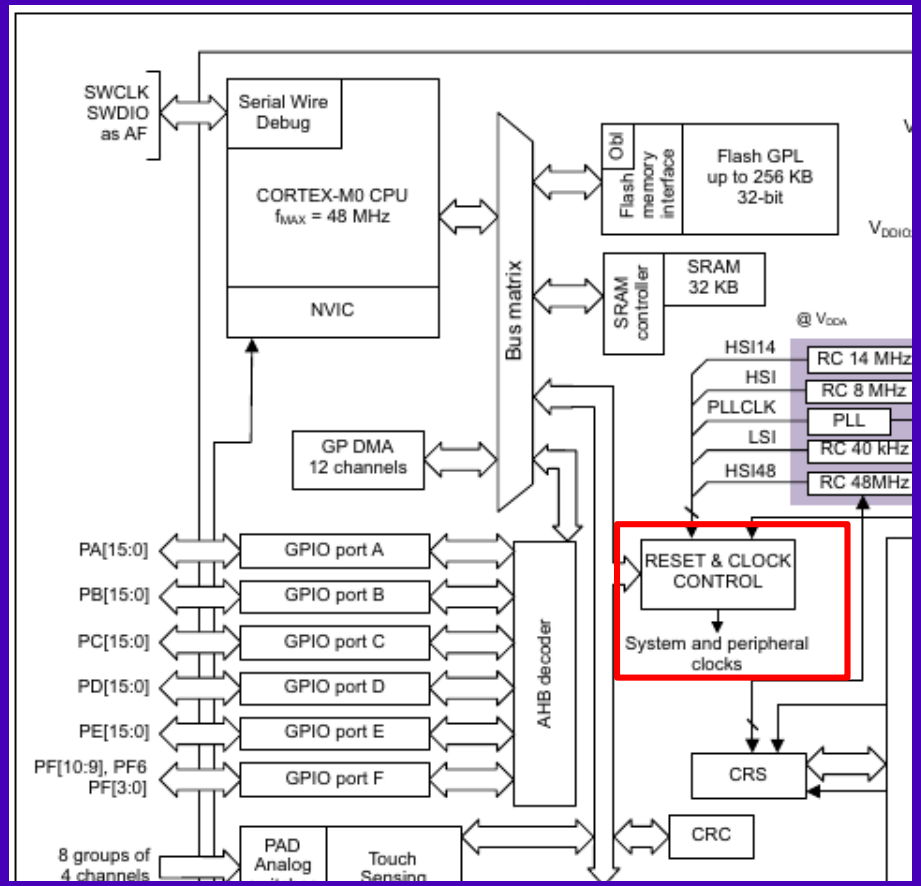
The STM32F091



- CPU
- Flash ROM
- SRAM
- Many kinds of built-in peripheral devices.
  - Connected by the AHB (Advanced High-Performance Bus)
  - Today, we look at the General Purpose I/O (GPIO) system and the RCC (Reset and Clock Control).

Figure 1 (page 12) in STM32F091xBC Datasheet

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.
    - 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.
  - Not the same thing as CPU registers.
  - 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.

		Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	IOPFEN	IOPEEN	USART4RST	IOPCEN	IOPBEN	IOPAEN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRCEN	Res.	FLITFEN	Res.	SRAMEN	DAM2EN	DMAEN
0x14	RCC_AHBENR																															
	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 long desc.

- 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

Enabling Ports

- 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.

		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
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

‘OR’ing values into ports

- No problem with reading all the bits of a control register, and writing back the same value.
- Also OK to read the value, turn on an additional bit in that value, and then write the new value back. That way, we won’t turn anything off.

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.	Res.	CRCEN	Res.	FLITFEN	Res.	SRAMEN	DAM2EN	DMAEN
	Reset value								0		0	0	0	0	0	0											0		1		1	0	0

Example code to enable Port C

```
.equ RCC,    0x40021000 // Base address of RCC control registers
.equ AHBENR, 0x14       // Offset of the RCC_AHBENR
.equ IOPCEN, 0x80000    // Value to enable “IO Port C”

ldr  r0, =RCC           // Load, into RO, base address of RCC
ldr  r1, [r0, #AHBENR]  // Load, into R1, value of RCC offset by AHBENR
ldr  r2, =IOPCEN        // Load, into R2, value to enable Port C
orrs r1, r2             // Combine R2 into R1
str  r1, [r0, #AHBENR]  // Store new bits into RCC_AHBENR
```

OK, now we can start doing I/O?

- No. Next, we must configure Port C.
- 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

Configure for 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.

Example Code to Enable Port C as Output

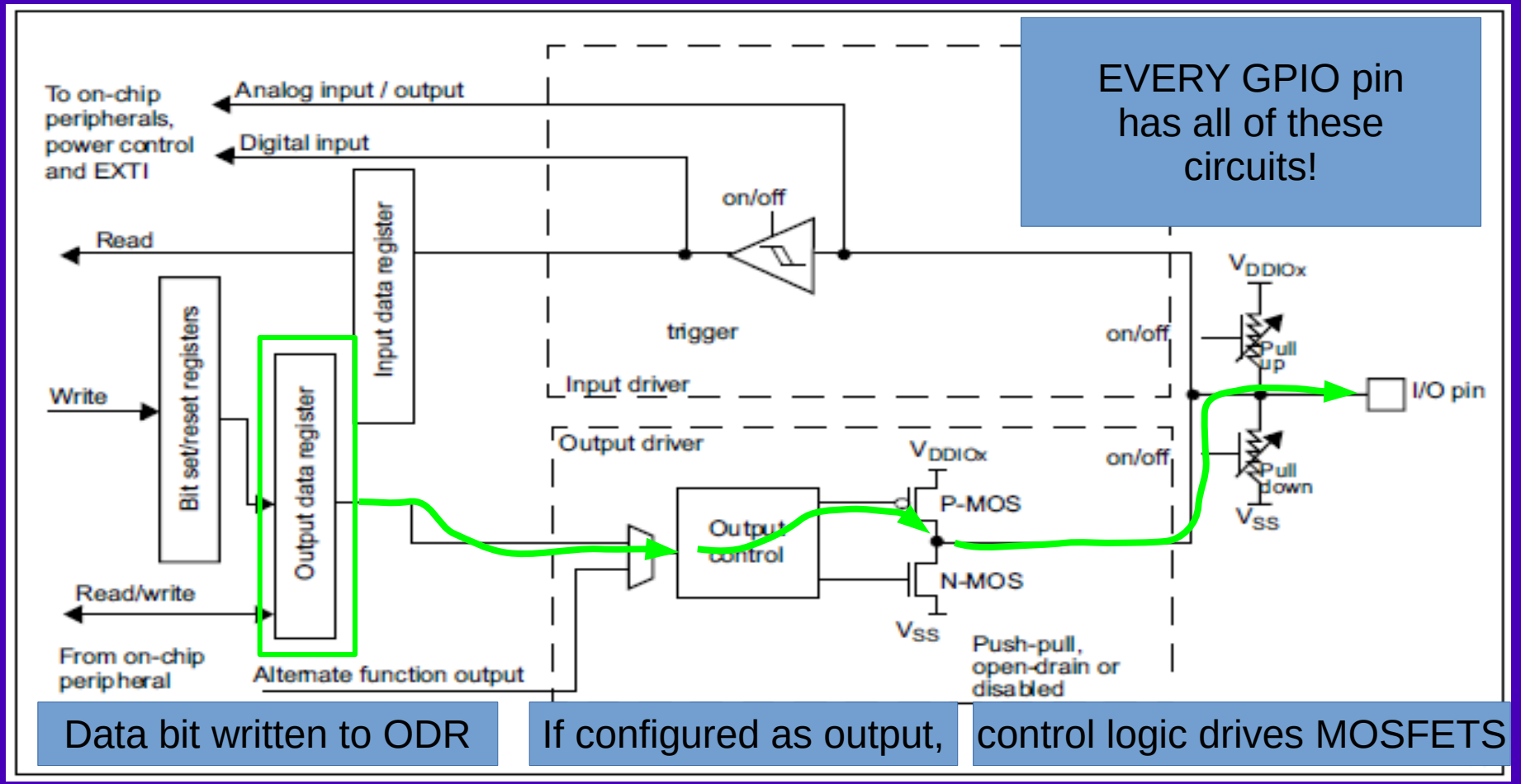
```
.equ GPIOC,  0x48000800 // Base address of GPIOC control registers
.equ MODER,  0x00       // Offset of the GPIOC_MODER
.equ OUT0_12,0x01555555 // Value to enable Port C bits 0 - 12

ldr  r0, =GPIOC         // Load, into R0, base address of GPIOC
ldr  r1, [r0, #MODER]   // Load, into R1, value of GPIOC offset by MODER
ldr  r2, =OUT0_12       // Load, into R2, value to enable PC0 … PC12
orrs r1, r2             // Combine R2 into R1
str  r1, [r0, #MODER]   // Store new bits into GPIOC_MODER
```

Now can we start doing I/O?

- Yes. Well…, at least, you can start doing *output*.
- 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.

What does the circuitry look like?



Example code to turn bits on/off

```
// After configuring RCC AHBENR and PORTC MODER, then we can...
.equ GPIOC, 0x48000800  // Base address of GPIOC control registers
.equ ODR, 0x14          // Offset of the GPIOC_MODER
.equ PC8, 0x100         // Port C pin 8
.equ PC89, 0x300        // Port C pins 8 and 9

    ldr  r3, =GPIOC     // Load, into R3, base address of GPIOC
    ldr  r2, =PC8       // The bit to turn on PC8
    ldr  r1, =PC89      // Bits to toggle PC8 and PC9
again:
    str  r2, [r3, #ODR] // Store bit pattern into ODR
    ldr  r0, =1000000000// Load one billion
    bl   nano_wait      // Wait one billion nanoseconds
    ldr  r2, [r3, #ODR] // Get the last value written to ODR
    eors r2, r1         // XOR it to toggle PC8 and PC9
    b    again          // Do it again
nano_wait:
    subs r0,#83
    bgt  nano_wait
    bx   lr
```

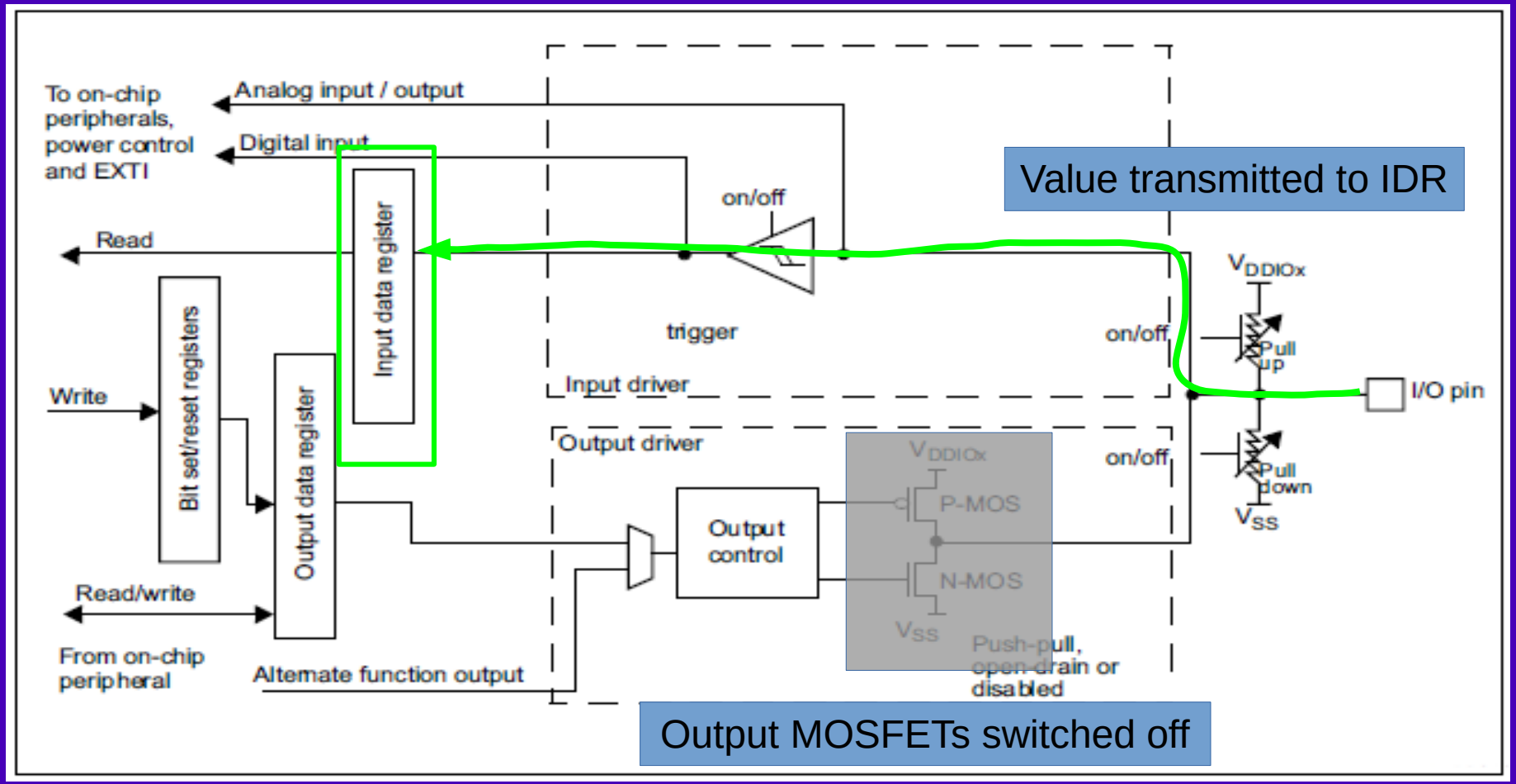
How about input?

- We set the Port's MODER register for '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

Input Circuit Path



Let's set Port A to be 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 code to set Port A pins as inputs

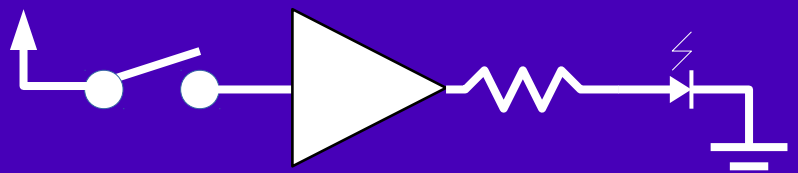
```
.equ GPIOA,  0x48000000 // Base address of GPIOA control registers
.equ MODER,  0x00       // Offset of the GPIOA_MODER
.equ CLR0_12,0x03ffffff // Value to clear from MODER

ldr  r0, =GPIOA         // Load, into RO, base address of GPIOA
ldr  r1, [r0, #MODER]   // Load, into R1, value of GPIOA offset by MODER
ldr  r2, =CLR0_12       // Load, into R2, value to clear
bics r1, r2             // Clear the bits in R1 that are set in R2
str  r1, [r0, #MODER]   // Store new bits into GPIOA_MODER
```

Example code to copy PA0 to PC8

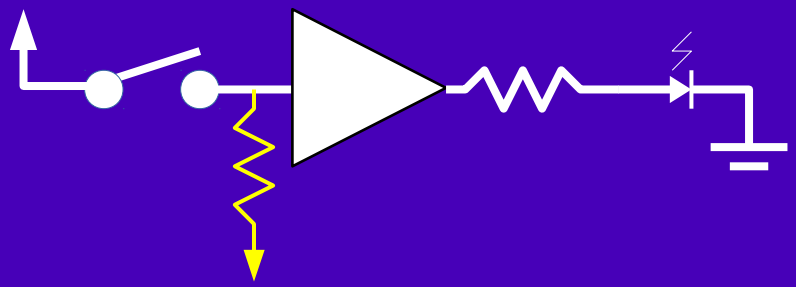
```
.equ GPIOA, 0x48000000  // Base address of GPIOA control registers
.equ GPIOC, 0x48000800  // Base address of GPIOC control registers
.equ IDR, 0x10          // Offset of the IDR
.equ ODR, 0x14          // Offset of the ODR

    ldr  r0, =GPIOA     // Load, into RO, base address of GPIOA
    ldr  r1, =GPIOC     // Load, into R1, base address of GPIOC
    movs r3, #1
again:
    ldr  r2, [r0, #IDR] // Load, into R2, value of GPIOA_IDR
    ands r2, r3         // Look at only bit zero.
    lsls r2, #8         // Translate bit 0 to bit 8
    str  r2, [r1, #ODR] // Store value to GPIOC_ODR
    b    again
```

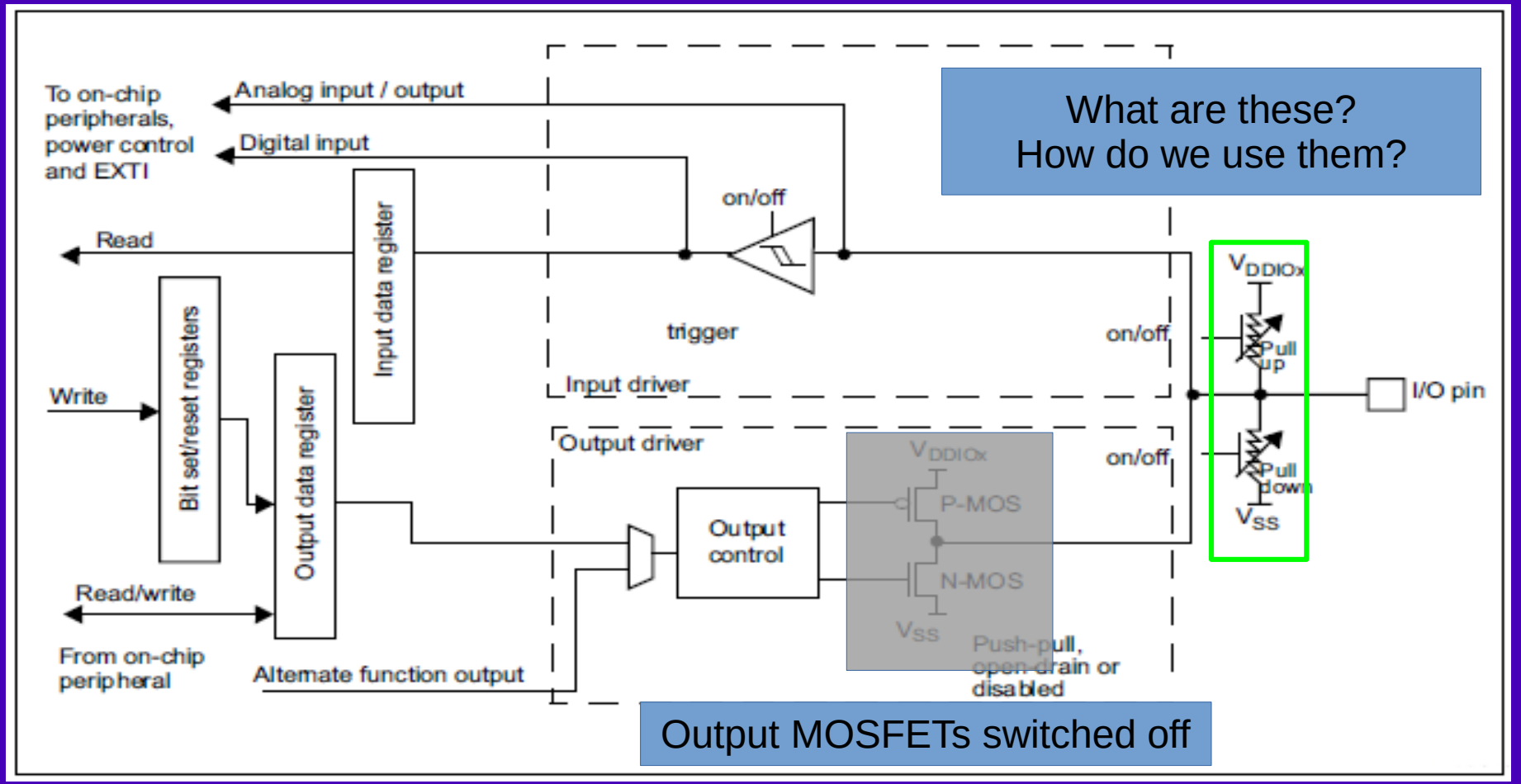


Electrical Problem: Input Stays High

- 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.

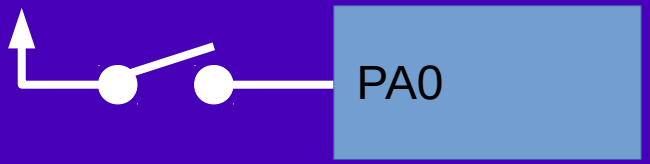


Input Circuit Path (again)



Pull-up / Pull-down resistors

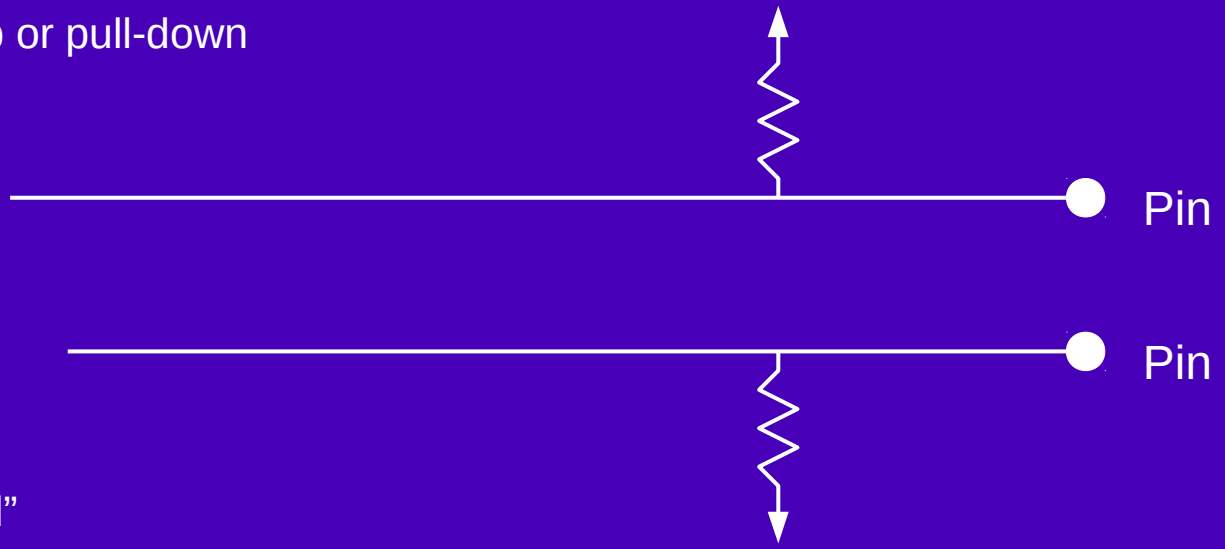
- Gently pull the input high or low in cases where it is not being driven. E.g.:



- 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”

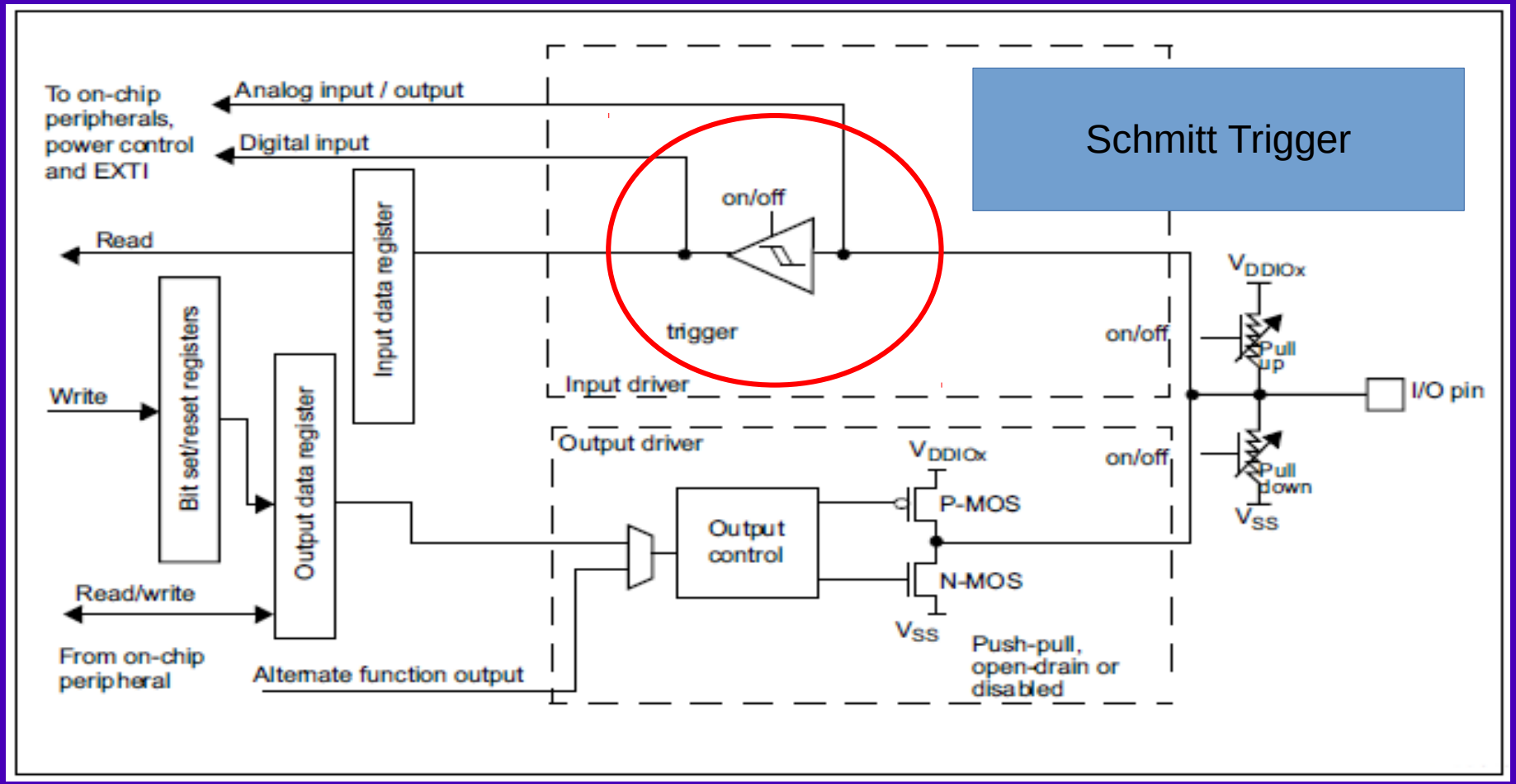


- How big are these resistors?
  - STM32F091xBC datasheet, Table 53, Pg 80: 25-55kΩ

Steps To Read SW2 (PA0)

- 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.

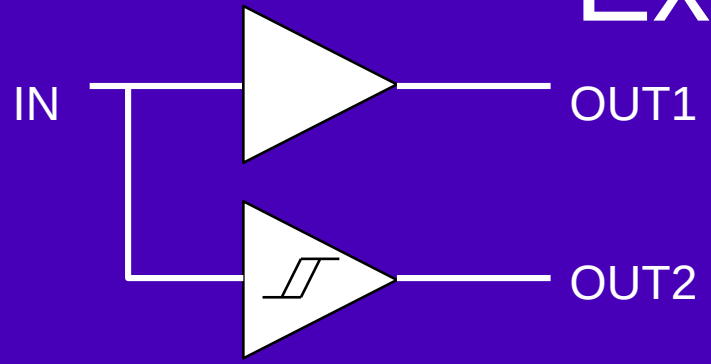
What is this on the input path?



What is a Schmitt Trigger?

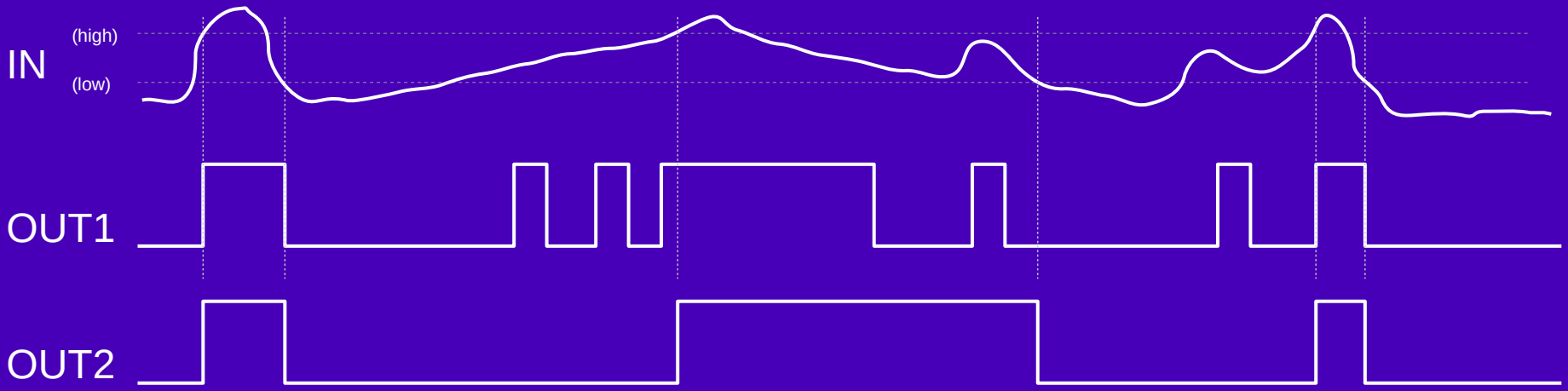
- Wikipedia says: Invented by American scientist Otto H. Schmitt in 1934 while he was a graduate student.
- It is a non-inverting buffer that is immune from problems with metastability for slowly-changing input values.
  - You know that logic gates have a maximum speed at which they can respond.
  - 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.

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.



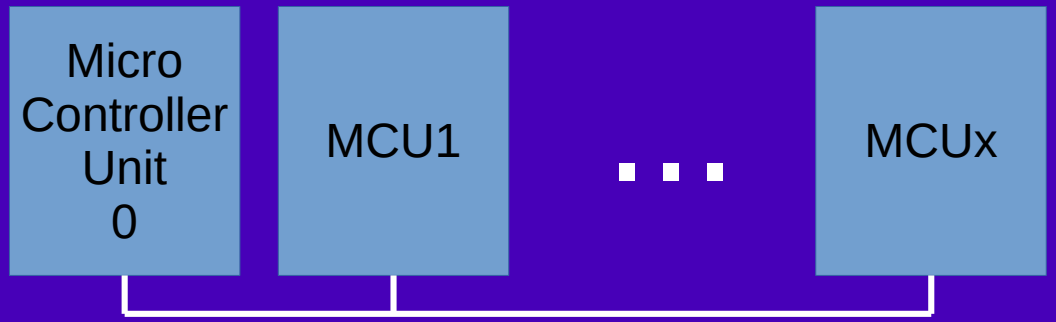
More about inputs

- 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. Pin is pulled low, but only floats high.

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^2C works.

Other GPIO registers: OSPEEDR

- 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

A ‘1’ in these bits will reset (turn off) an ODR bit.

A ‘1’ in these bits will set (turn on) an ODR bit.

		BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
0x18	**GPIOx_BSRR** (where x = **A..F**)	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
	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	0	0
0x28	**GPIOx_BRR** (where x = **A..F**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A ‘1’ in these bits will reset (turn off) an ODR bit.

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.

Reading assignment:
STM32F0x1 Family Reference, Chapter 12, pages 217 – 228, "Interrupts and events"

Examples of BRR / BSRR

```
// Assume current ODR is
//      (0xF0F0)   1111000011110000
ldr r0, =GPIOC
ldr r1, =0x3113 // 0011000100010011
str r1, [r0, #BRR]


// New value of ODR is
//                 1100000011100000
```

```
// Assume current ODR is
//           (0xF0F0) 1111000011110000
ldr r0, =GPIOC
ldr r1, =0x30100302 (-0011000000010000)
                    (+0000001100000010)
str r1, [r0, #BSRR]

// New value of ODR is
//           (0xC3E2) 1100001111100010
```

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

Other GPIO registers: LCKR

- LCKR: (Lock Register)
  - Write a sequence of values to LCKR and it will no longer allow you to change GPIOx control registers until the next system reset.
  - Probably don’t need this for ECE 362.
  - Use this if you need safety-critical controls.
    - Like ECE 477 when you make flying robot flamethrowers.

Other GPIO registers: AFRx

- AFRL, AFRH: Alternate Function Register (hi/lo)
  - Configures a port pins for a pin-specific function.
  - 2 32-bit registers. 4 bits per pin. 8 pins per register.
  - See STM32F091xBC datasheet, Table 14 – 19, pp 41 – 44.

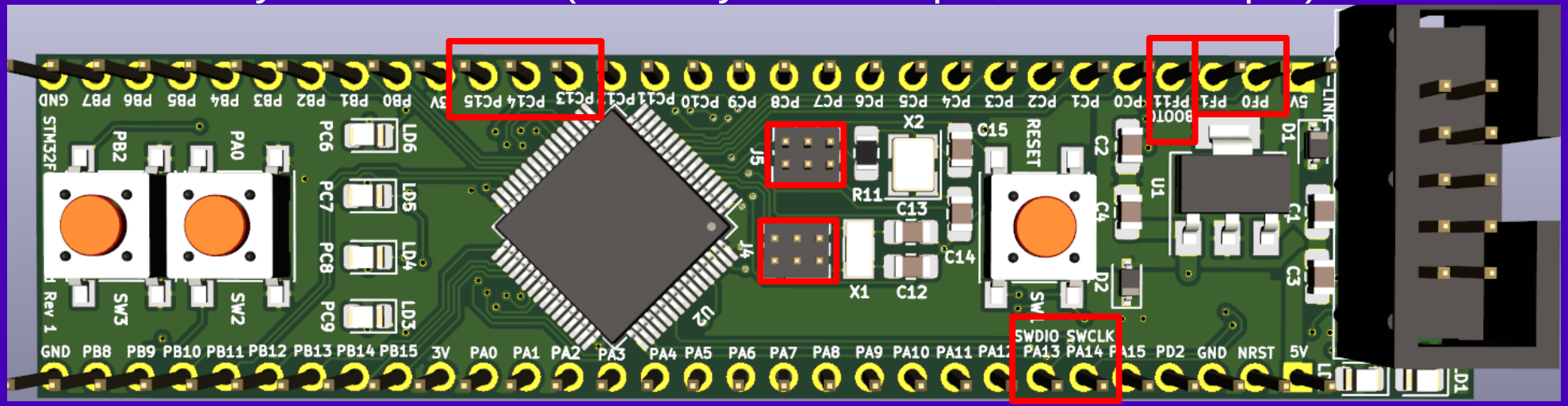
- We’ll see more about this in days to come when we look at other types of I/O and peripherals.

What can we do with GPIO

- We often use GPIO to mimic bus protocols.
  - This is typically referred to as "bit banging."
  - Usually slower than a native interface.
  - e.g. use some pins to strobe address lines and other pins to read and write a bidirectional data bus, and a few more for control signals like read, write, or enable.
    - Instant external RAM controller.
    - 44 bits of usable GPIO. 2 control lines, 8 data, 34 address.
  - e.g. 8 bits data, 2 control lines, 2 select lines: LCD controller
  - e.g. one pin for output, one pin for input, simple serial connection.
  - e.g. MISO, MOSI, SS, SCK: SDcard interface.

Do Not Use The Following Pins

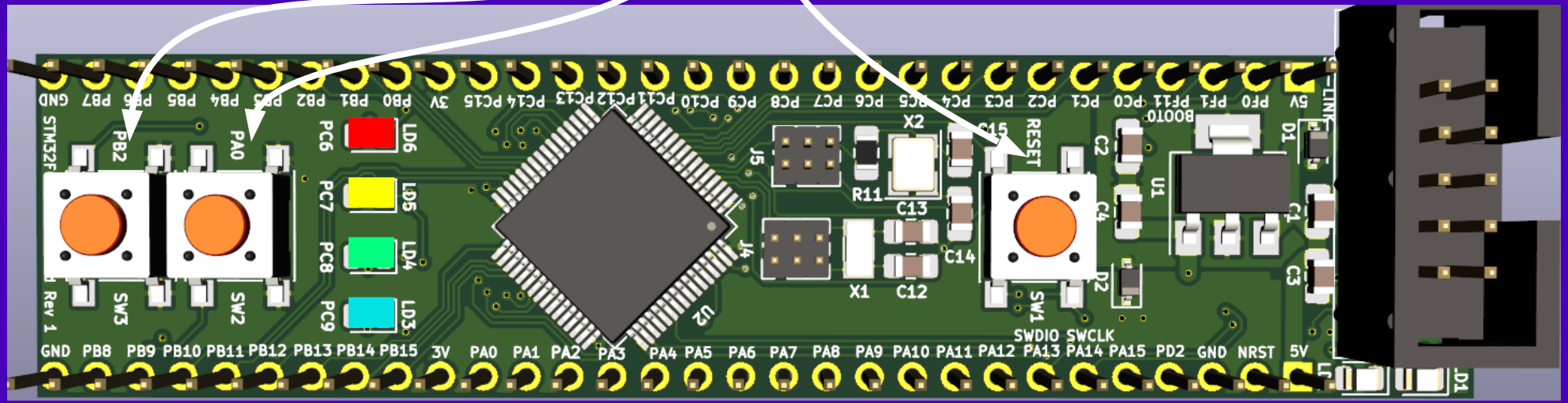
- External oscillator pins:
  - PC13, PC14, PC15: 32.768 kHz crystal (configured through J4)
  - PF0, PF1: 8MHz crystal (configured through J5)
- PA13, PA14: Programming Pins
- PF11: System Boot Pin (use only as an output, not as an input)



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.