

# Lecture 04

# Input and Output

CE346 – Microprocessor System Design  
Branden Ghena – Fall 2021

Some slides borrowed from:  
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

# Administrivia

- Office Hours

- My office hours are: Fridays 1-2pm on Zoom (see Canvas homepage)
- Also available by request (make a post to Campuswire)
  
- General lab office hours will hopefully be posted this weekend?

- Lab tomorrow!

- Tech CG50
- <https://www.mccormick.northwestern.edu/contact/tech-room-finder-map.php?id=CG50&room-floor=0&room-id=259&room-ingress=>

# General Lab access

- We'll double-check on Friday, but your ID cards should get you into CG50 at any time
  - You are welcome to go in there to work on stuff 24/7
- However, Tech is doing contact tracing and needs your help
  - If you are going to lab (apart from class/office hours) please send an email to [ece-labs@northwestern.edu](mailto:ece-labs@northwestern.edu) 24 hours in advance
  - You're not scheduling anything, and they likely won't respond, but they'll keep the info for Tech for contact-tracing purposes
  - If you don't remember 24 hours in advance:
    - 2 hours, 15 minutes, 0 hours, or -1 hours are all still helpful

# Today's Goals

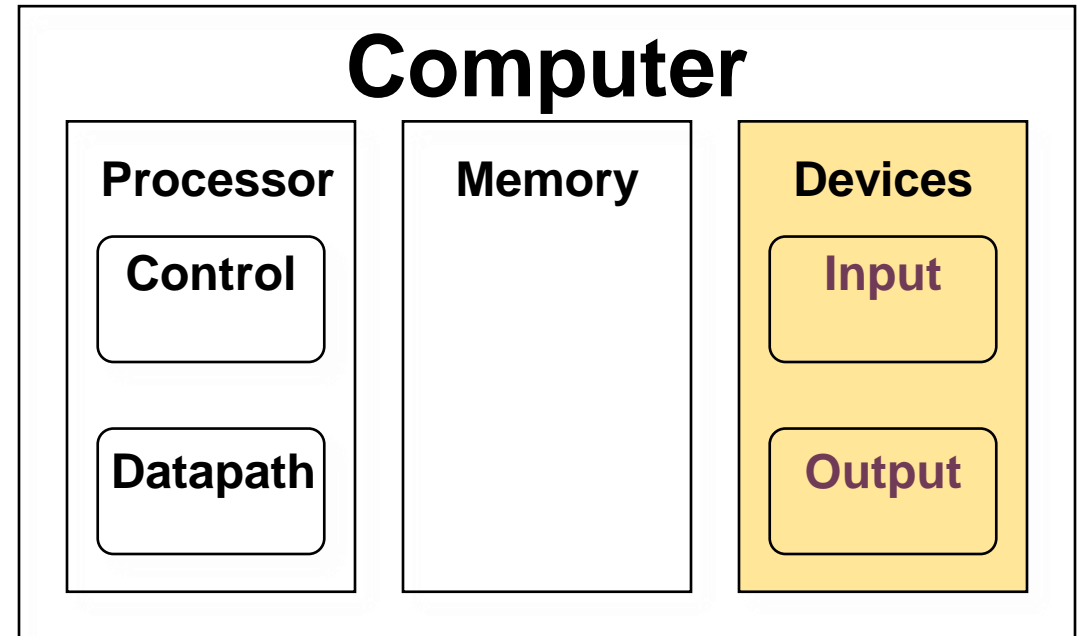
- How does a microcontroller interact with peripherals to perform input and output operations?
  - Memory-Mapped I/O
  - Interrupts
  - DMA
- Explore reliable use of MMIO
- Discuss interaction patterns for Interrupts and DMA
- Understand General Purpose I/O (GPIO) peripherals
  - And what kind of configurations they have

# Outline

- **I/O Motivation**
- Memory-Mapped I/O
- Interrupts
- Controlling digital signals
  - GPIO
  - GPIOTE
- DMA

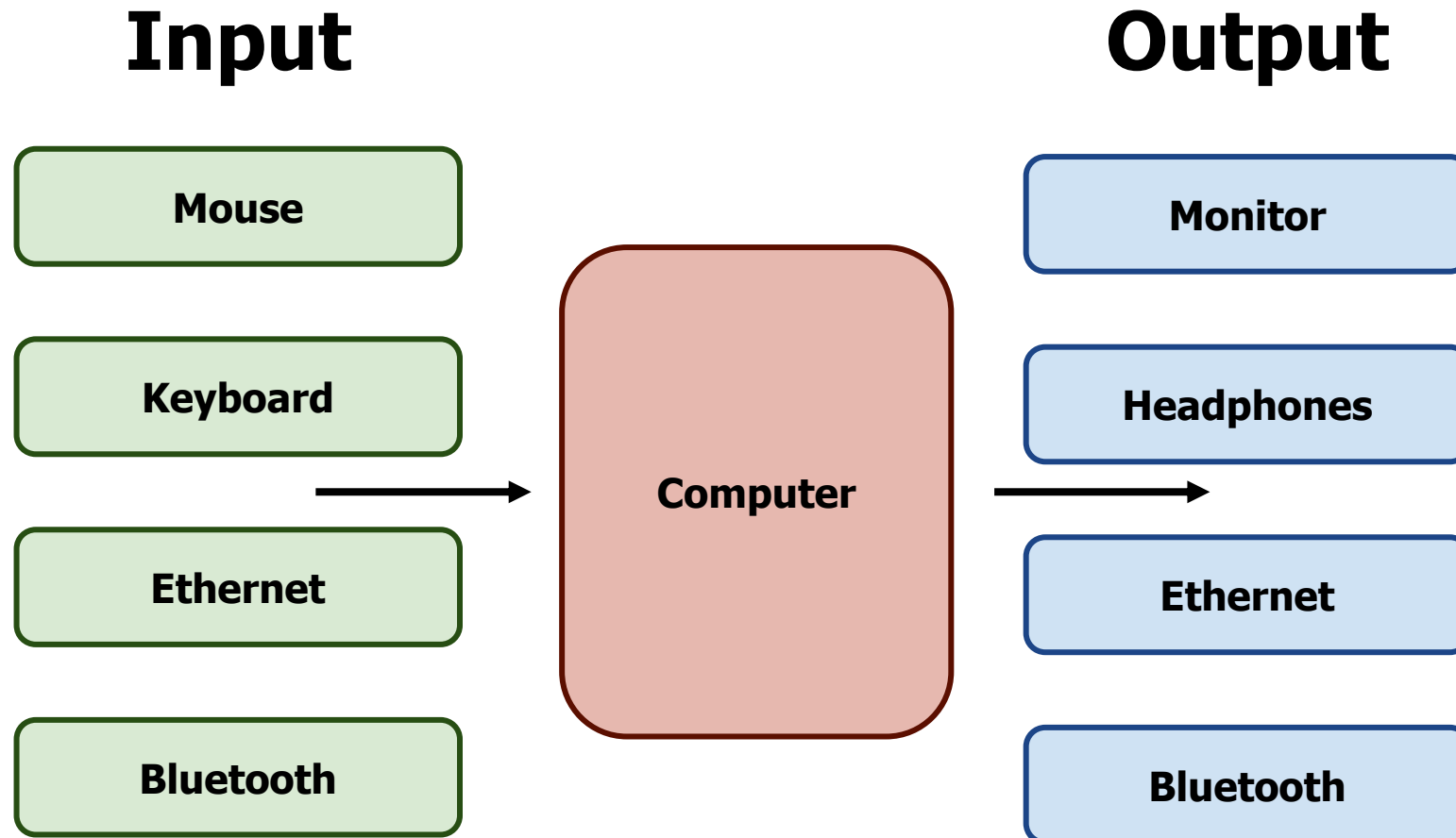
# Devices are the point of computers

- Traditional systems need to receive input from users and output responses
  - Keyboard/mouse
  - Disk
  - Network
  - Graphics
  - Audio
  - Various USB devices

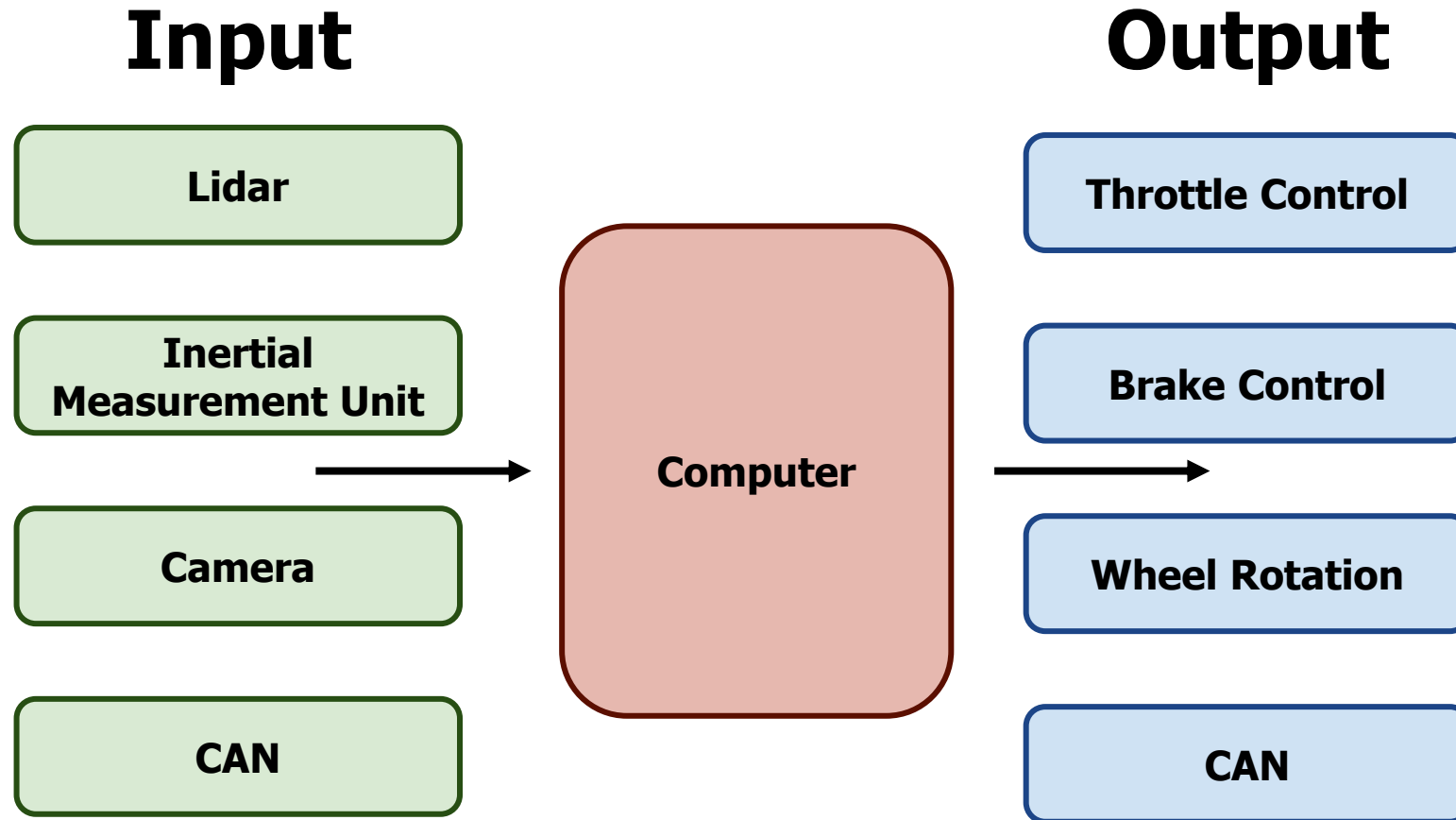


- Embedded systems have the same requirement, just more types of IO

# Devices are core to useful general-purpose computing



# Devices are essential to cyber-physical systems too





# Device access rates vary by many orders of magnitude

- Rates in bit/sec

Device	Behavior	Partner	Data Rate (Kb/s)
Keyboard	Input	Human	0.2
Mouse	Input	Human	0.4
Microphone	Output	Human	700.0
Bluetooth	Input or Output	Machine	20,000.0
Hard disk drive	Storage	Machine	100,000.0
Wireless network	Input or Output	Machine	300,000.0
Solid state drive	Storage	Machine	500,000.0
Wired LAN network	Input or Output	Machine	1,000,000.0
Graphics display	Output	Human	3,000,000.0

- System must be able to handle each of these

- Sometimes needs low overhead
- Sometimes needs to not wait around

# Outline

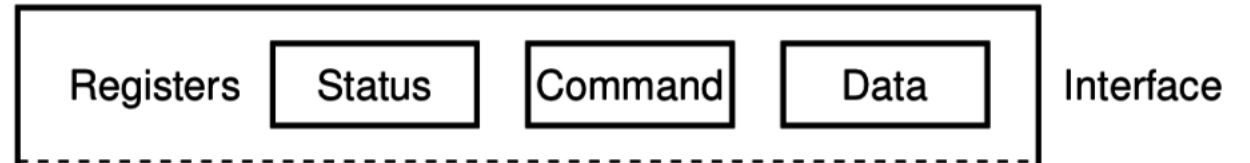
- I/O Motivation
- **Memory-Mapped I/O**
- Interrupts
- Controlling digital signals
  - GPIO
  - GPIOTE
- DMA

# How does a computer talk with peripherals?

- A peripheral is a hardware unit within a microcontroller
  - Sort of a “computer-within-the-computer”
  - Performs some kind of action given input, generates output
- We interact with a peripheral’s interface
  - Called registers (actually are from EE perspective, but you can’t use them)
  - Read/Write like they’re data

## • How do we read/write them?

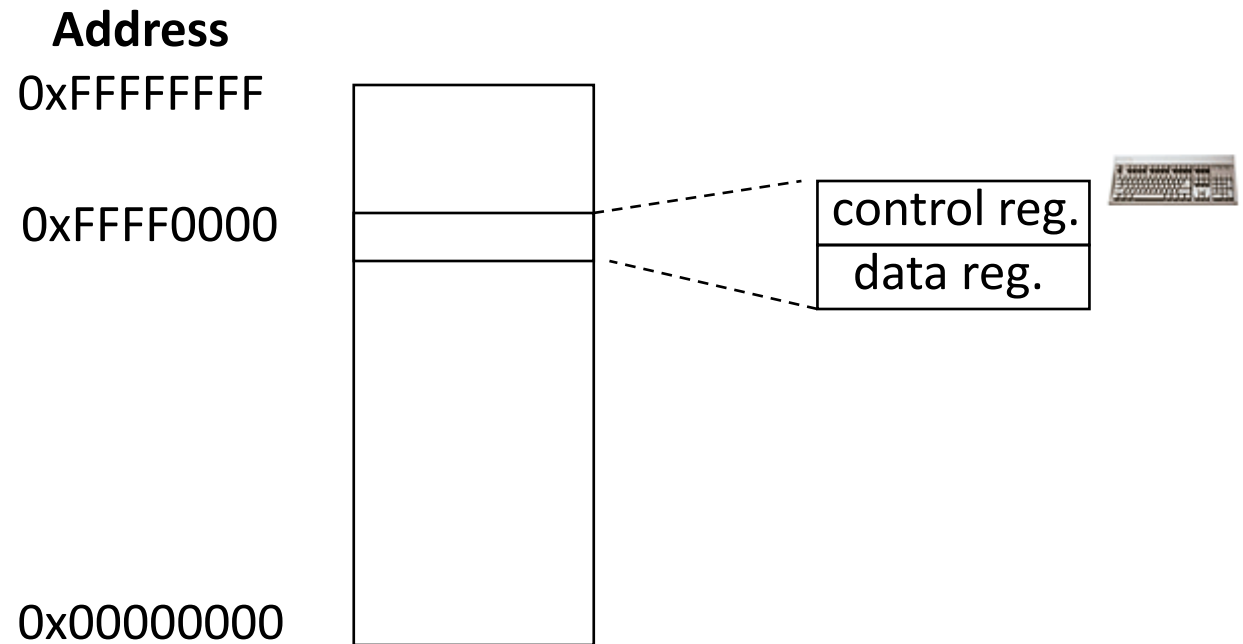
- Options:
  - Special assembly instructions
  - Treat like normal memory



# Memory-mapped I/O (MMIO): treat devices like normal memory

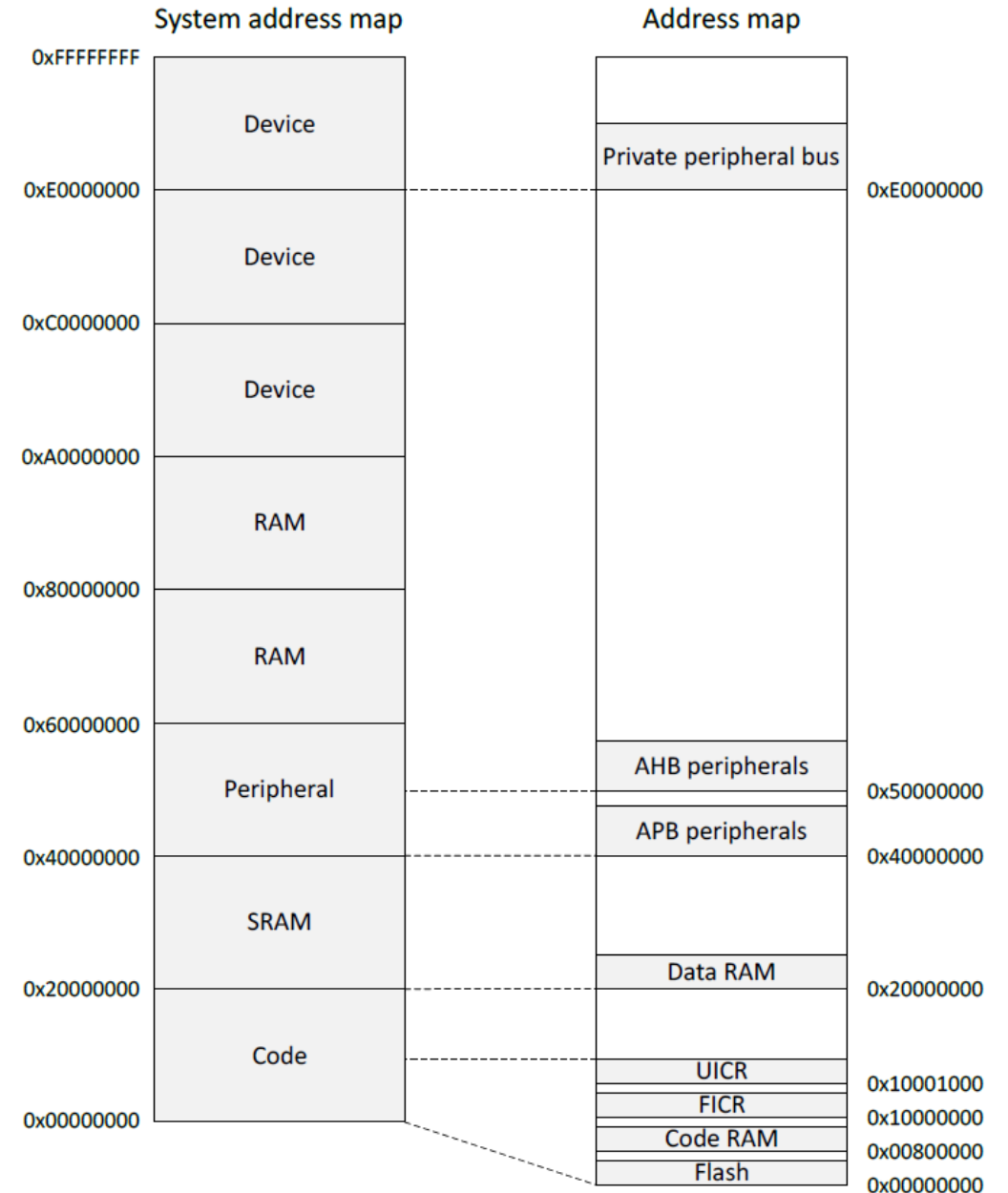
- Certain physical addresses do not actually go to memory
- Instead they correspond to peripherals
  - And any instruction that accesses memory can access them too!

- Every microcontroller I've ever seen uses MMIO



# Memory map on nRF52833

- Flash 0x00000000
- SRAM 0x20000000
- APB peripherals 0x40000000
  - Everything but GPIO
- AHB peripherals 0x50000000
  - Just GPIO
- UICR – User Information Config
- FICR – Factory Information Config



# Example nRF52 peripheral placement

- 0x1000 is plenty of space for each peripheral
  - 1024 registers, each 32 bits
  - No reason to pack them tighter than that

5	0x40005000	NFCT	NFCT	Near field communication tag
6	0x40006000	GPIOTE	GPIOTE	GPIO tasks and events
7	0x40007000	SAADC	SAADC	Analog to digital converter
8	0x40008000	TIMER	TIMER0	Timer 0
9	0x40009000	TIMER	TIMER1	Timer 1
10	0x4000A000	TIMER	TIMER2	Timer 2
11	0x4000B000	RTC	RTC0	Real-time counter 0
12	0x4000C000	TEMP	TEMP	Temperature sensor
13	0x4000D000	RNG	RNG	Random number generator
14	0x4000E000	ECB	ECB	AES electronic code book (ECB) mode block encryption
15	0x4000F000	AAR	AAR	Accelerated address resolver

# TEMP on nRF52833 example

- Internal temperature sensor
  - 0.25° C resolution
  - Range equivalent to microcontroller IC (-40° to 105° C)
  - Various configurations for the temperature conversion (ignoring)

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

*Table 110: Instances*

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

# MMIO addresses for TEMP

- What addresses do we need? (ignore interrupts for now)
  - 0x4000C000 – TASKS\_START
  - 0x4000C100 – EVENTS\_DATARDY
  - 0x4000C508 - TEMP

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

*Table 110: Instances*

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)



# Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

# Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

- 0x4000C000 is cast to a uint32\_t\*
- Then dereferenced
- And we write 1 to it
  
- “There are 32-bits of memory at 0x4000C000. Write a 1 there.”

# Example code

- To the terminal!

# Example code

```
// loop forever
while (1) {

    // start a measurement
    *(uint32_t*)(0x4000C000) = 1;

    // wait until ready
    volatile uint32_t ready = *(uint32_t*)(0x4000C100);
    while (!ready) {
        ready = *(uint32_t*)(0x4000C100);
    }

    /* WARNING: we can't write the code this way!
     * Without `volatile`, the compiler optimizes out the memory access
     while (!*(uint32_t*)(0x4000C100));
     */

    // read data and print it
    volatile int32_t value = *(int32_t*)(0x4000C508);
    float temperature = ((float)value)/4.0;
    printf("Temperature=%f degrees C\n", temperature);

    nrf_delay_ms(1000);
}
```

# Using structs to manage MMIO access

- Writing simple C code and access peripherals is great!
- Problems:
  - Need to remember all these long addresses
  - Need to make sure compiler doesn't stop us!
- Solution:
  - Wrap entire access in a struct!
  - Compilers turn it into the same thing in the end anyways

# C structs

- Collection of variables placed together in memory

```
typedef struct {  
    uint32_t variable_one;  
    uint32_t variable_two;  
    uint32_t array[2];  
} example_struct_t;
```

- Placement rules - Variables are placed adjacent to each other in memory except:
  - Variables are always placed at a multiple of their size
  - Padding added to the end to make the total size a multiple of the biggest member
- Microcontrollers can usually ignore these: all registers are the same size!

# Temperature peripheral MMIO struct

```
typedef struct {
```

```
} temp_regs_t;
```

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

# Temperature peripheral MMIO struct

```
typedef struct {
    uint32_t TASKS_START;
    uint32_t TASKS_STOP;
    uint32_t _unused_A[62];
    uint32_t EVENTS_DATARDY;
    uint32_t _unused_B[64+64+1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[64+64];
    uint32_t TEMP;
} temp_regs_t;
```

```
volatile temp_regs_t* TEMP_REGS = (temp_regs_t*) (0x4000C000);
```

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)



# Temperature peripheral MMIO struct

```
typedef struct {
    uint32_t TASKS_START;
    uint32_t TASKS_STOP;
    uint32_t _unused_A[62];
    uint32_t EVENTS_DATARDY;
    uint32_t _unused_B[64+64+1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[64+64];
    uint32_t TEMP;
} temp_regs_t;
```

```
volatile temp_regs_t* TEMP_REGS = (temp_regs_t*) (0x4000C000);
```

```
// code to access
```

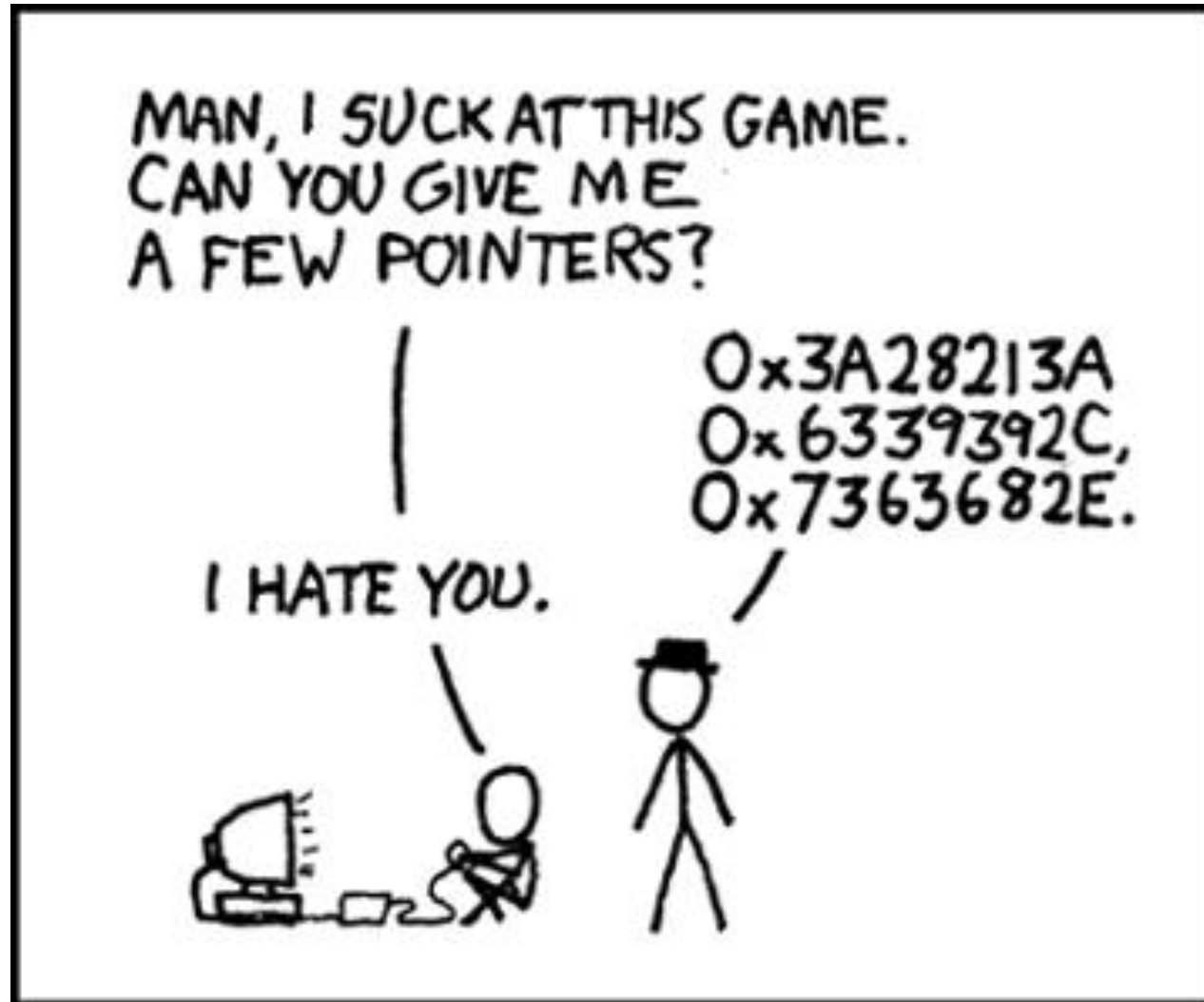
```
TEMP_REGS->TASKS_START = 1;
```

```
while (TEMP_REGS->EVENTS_DATARDY == 0);
```

```
float temperature = ((float)TEMP_REGS->TEMP) / 4.0;
```

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

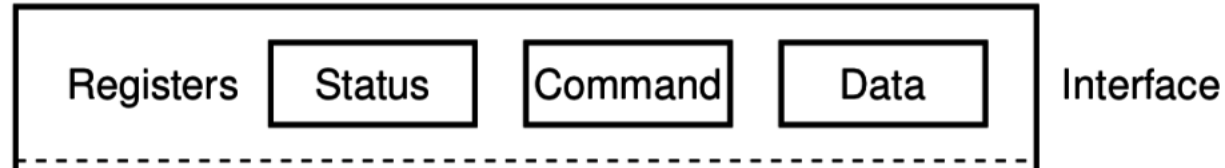
# Break + relevant xkcd



# Outline

- I/O Motivation
- Memory-Mapped I/O
- **Interrupts**
- Controlling digital signals
  - GPIO
  - GPIOTE
- DMA

# What do interactions with devices look like?



1. `while STATUS==BUSY; Wait`
  - (Need to make sure device is ready for a command)
2. Write value(s) to DATA
3. Write command(s) to COMMAND
4. `while STATUS==BUSY; Wait`
  - (Need to make sure device has completed the request)
5. Read value(s) from Data

This is the “polling” model of I/O.

“Poll” the peripheral in software repeatedly to see if it’s ready yet.

# Waiting can be a waste of CPU time

## **1. while STATUS==BUSY; Wait**

- **(Need to make sure device is ready for a command)**

2. Write value(s) to DATA

3. Write command(s) to COMMAND

## **4. while STATUS==BUSY; Wait**

- **(Need to make sure device has completed the request)**

5. Read value(s) from Data

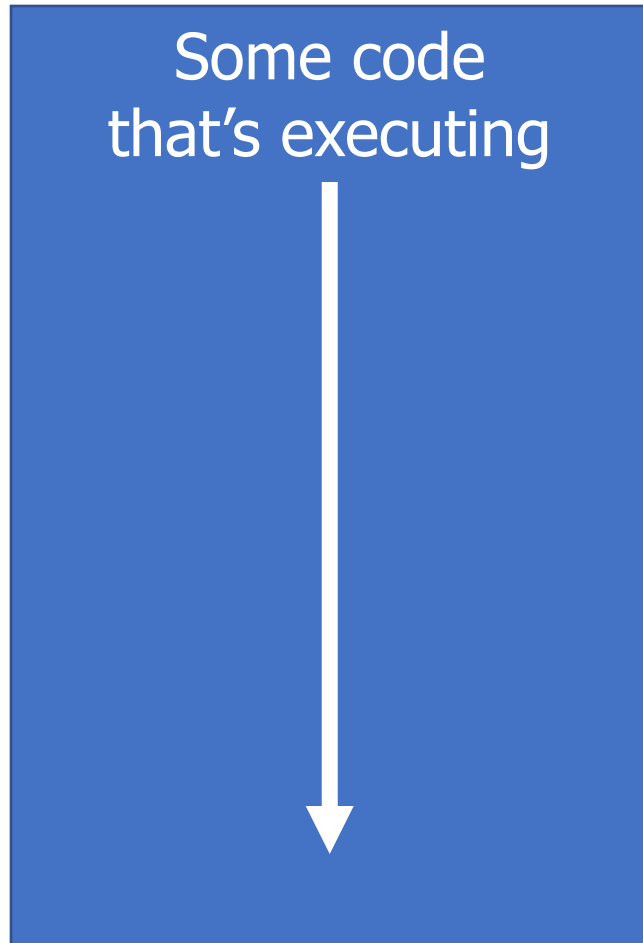
## • Imagine a keyboard device

- CPU could be waiting for minutes before data arrives
- Need a way to notify CPU when an event occurs
  - Interrupts!

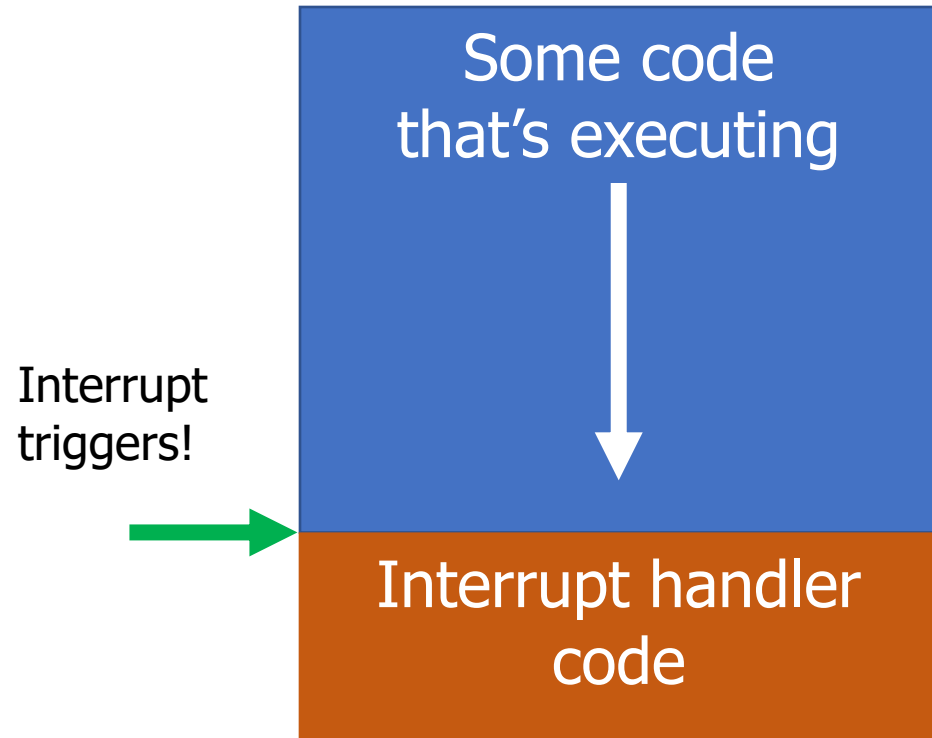
# Interrupts

- What is an interrupt?
  - Some event which causes the processor to stop normal execution
  - The processor instead jumps to a handler for that event
- What causes interrupts?
  - Hardware exceptions
    - Divide by zero, Undefined Instruction, Memory bus error
  - Software
    - Syscall, Software Interrupt (SWI)
  - External hardware
    - Input pin, Timer, various "Data Ready"

# Interrupts, visually

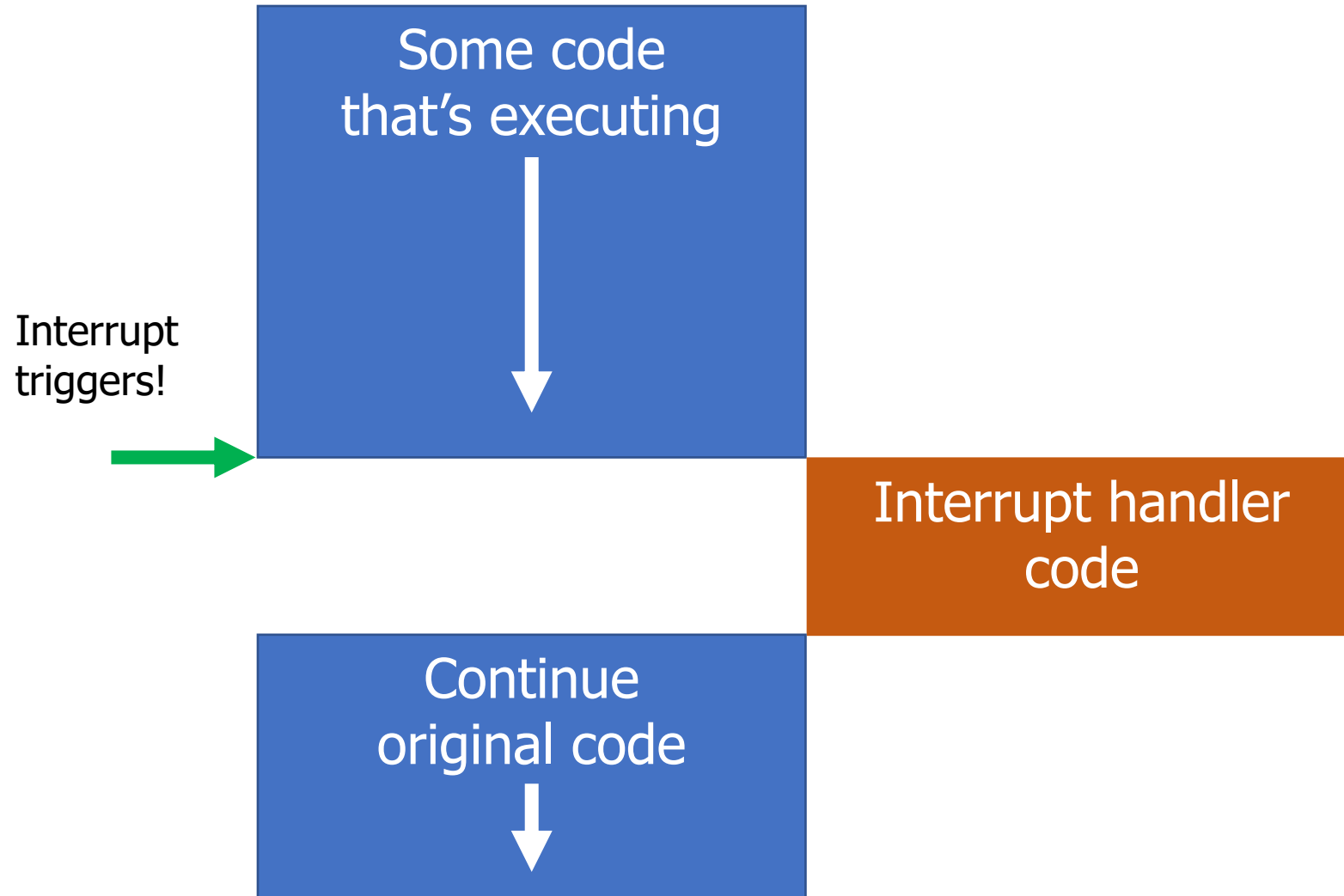


# Interrupts, visually





# Interrupts, visually



# ARM Nested Vectored Interrupt Controller (NVIC)

Interrupts can preempt other interrupts!

Jump directly to the interrupt handler

Handles interrupt entry and exit

- Stacking
- Unstacking
- Priorities

- Manages interrupt requests (IRQ)
  - Stores all callee saved registers on the stack
    - So the handler code doesn't overwrite them
  - Moves the PC to the proper handler, a.k.a. Interrupt Service Routine (ISR)
  - Restores registers after handler returns and moves PC back

# ARM Vector table

- List of function pointers to handler for each interrupt/exception
- First 15 are architecture-specific exceptions
- After that are microcontroller interrupt signals

**Table 7.1** List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Nonmaskable interrupt (external NMI input)
3	Hard fault	-1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called <i>prefetch abort</i> if it is an instruction fetch or <i>data abort</i> if it is a data access)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7-10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

**Table 7.2** List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
...	...	...
255	External Interrupt #239	Programmable

# Vector table in software

- Placed in its own section
  - LD file puts it first in Flash
- Reset\_Handler determines where software starts executing
- After that are all exception and interrupt handlers
  - All function pointers to some C code somewhere

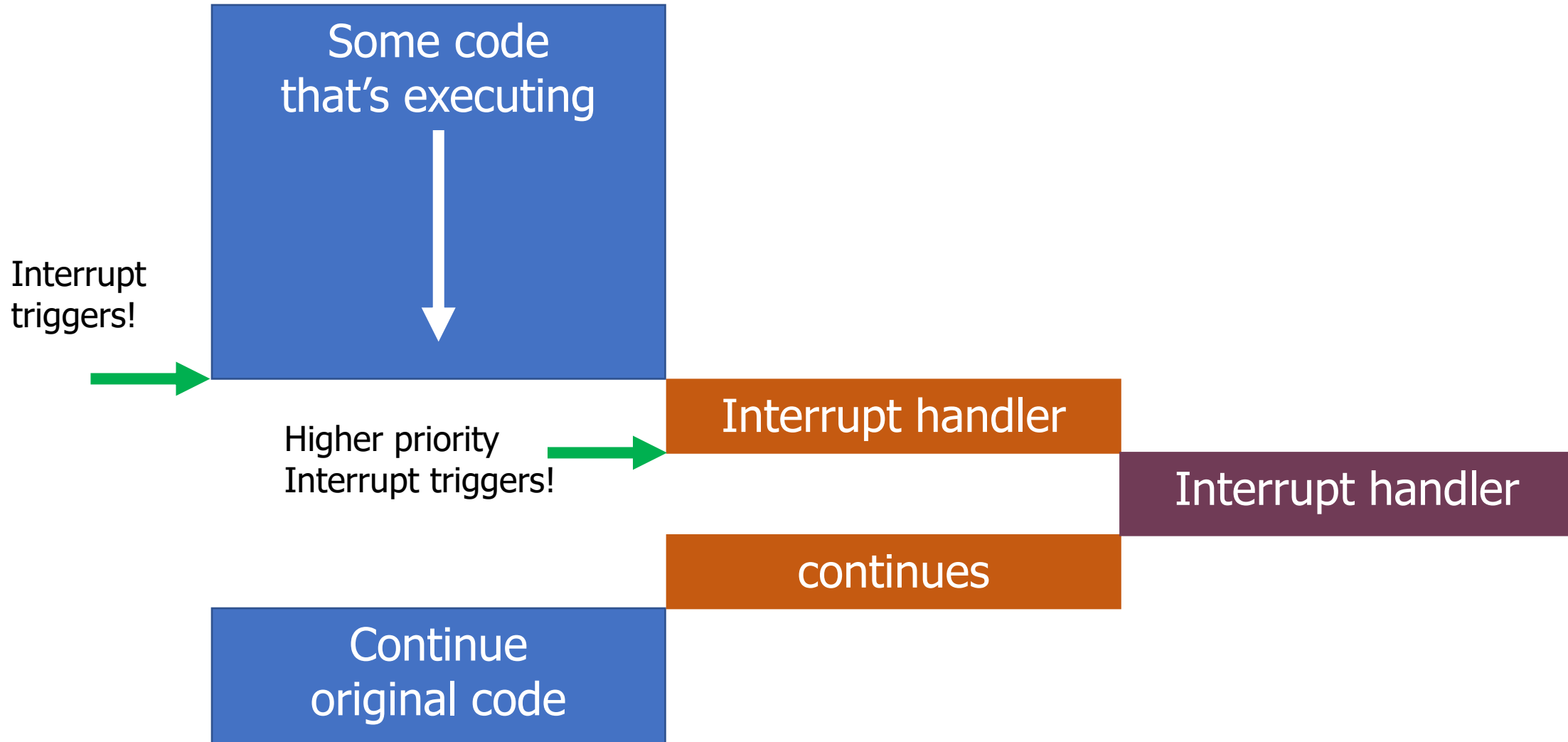
```
.section .isr_vector
.align 2
.globl __isr_vector
__isr_vector:
    .long    __StackTop           /* Top of Stack */
    .long    Reset_Handler
    .long    NMI_Handler
    .long    HardFault_Handler
    .long    MemoryManagement_Handler
    .long    BusFault_Handler
    .long    UsageFault_Handler
    .long    0                   /*Reserved */
    .long    0                   /*Reserved */
    .long    0                   /*Reserved */
    .long    0                   /*Reserved */
    .long    SVC_Handler
    .long    DebugMon_Handler
    .long    0                   /*Reserved */
    .long    PendSV_Handler
    .long    SysTick_Handler

    /* External Interrupts */
    .long    POWER_CLOCK_IRQHandler
    .long    RADIO_IRQHandler
    .long    UARTE0_UART0_IRQHandler
    .long    SPIM0_SPIS0_TWIM0_TWI0_IRQHandler
    .long    SPIM1_SPIS1_TWIM1_TWI1_IRQHandler
    .long    NFCT_IRQHandler
    .long    GPIOTE_IRQHandler
    .long    SAADC_IRQHandler
```

# NVIC functionality

- NVIC functions
  - NVIC\_EnableIRQ(number)
  - NVIC\_DisableIRQ(number)
  - NVIC\_SetPriority(number, priority)
    - Technically 256 priorities
    - Only 8 are implemented
- Must enable interrupts in two places!
  - Enabling interrupt in the peripheral will generate the signal
  - Enabling interrupt in the NVIC will cause signal to jump to handler
- Priority determines which interrupt goes first
  - And determines how interrupts are nested

# Nested interrupts, visually



# Break + Open Question

- When should a system use interrupts versus polling?

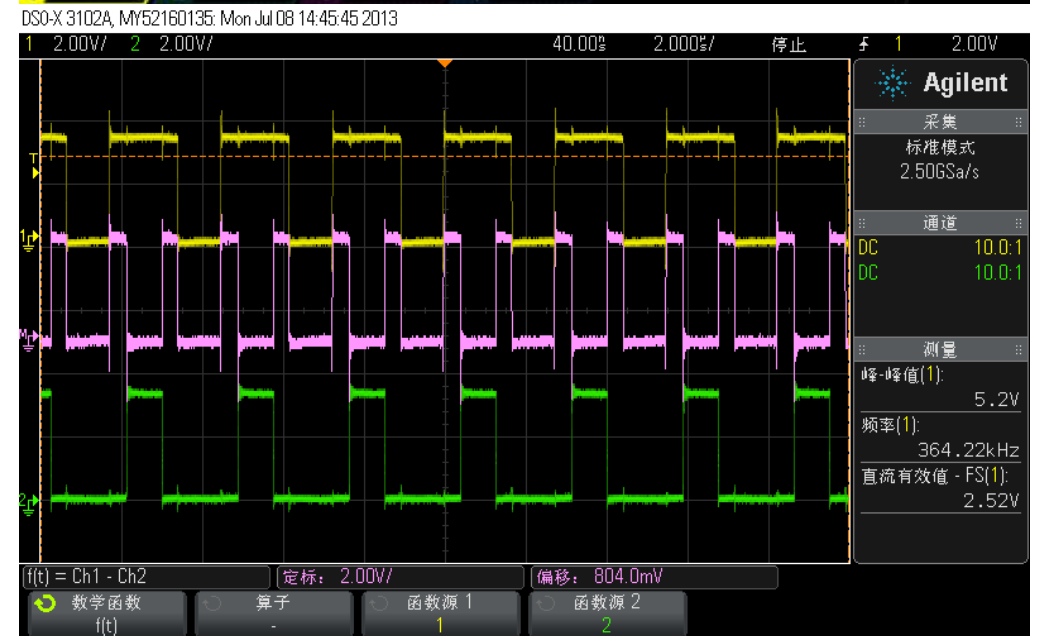
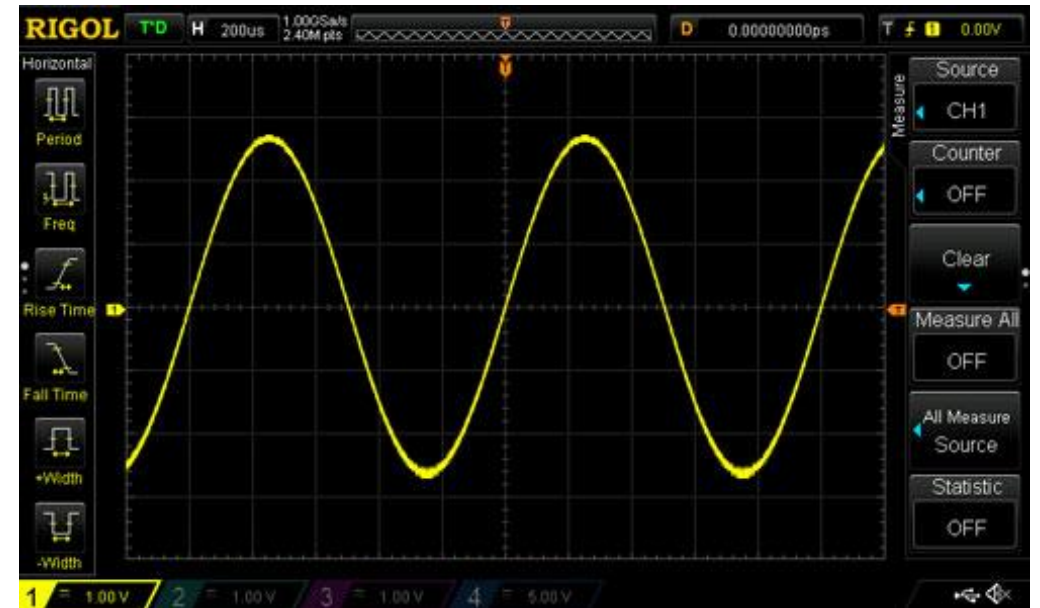
# Outline

- I/O Motivation
- Memory-Mapped I/O
- Interrupts
- **Controlling digital signals**
  - **GPIO**
  - GPIOTE
- DMA



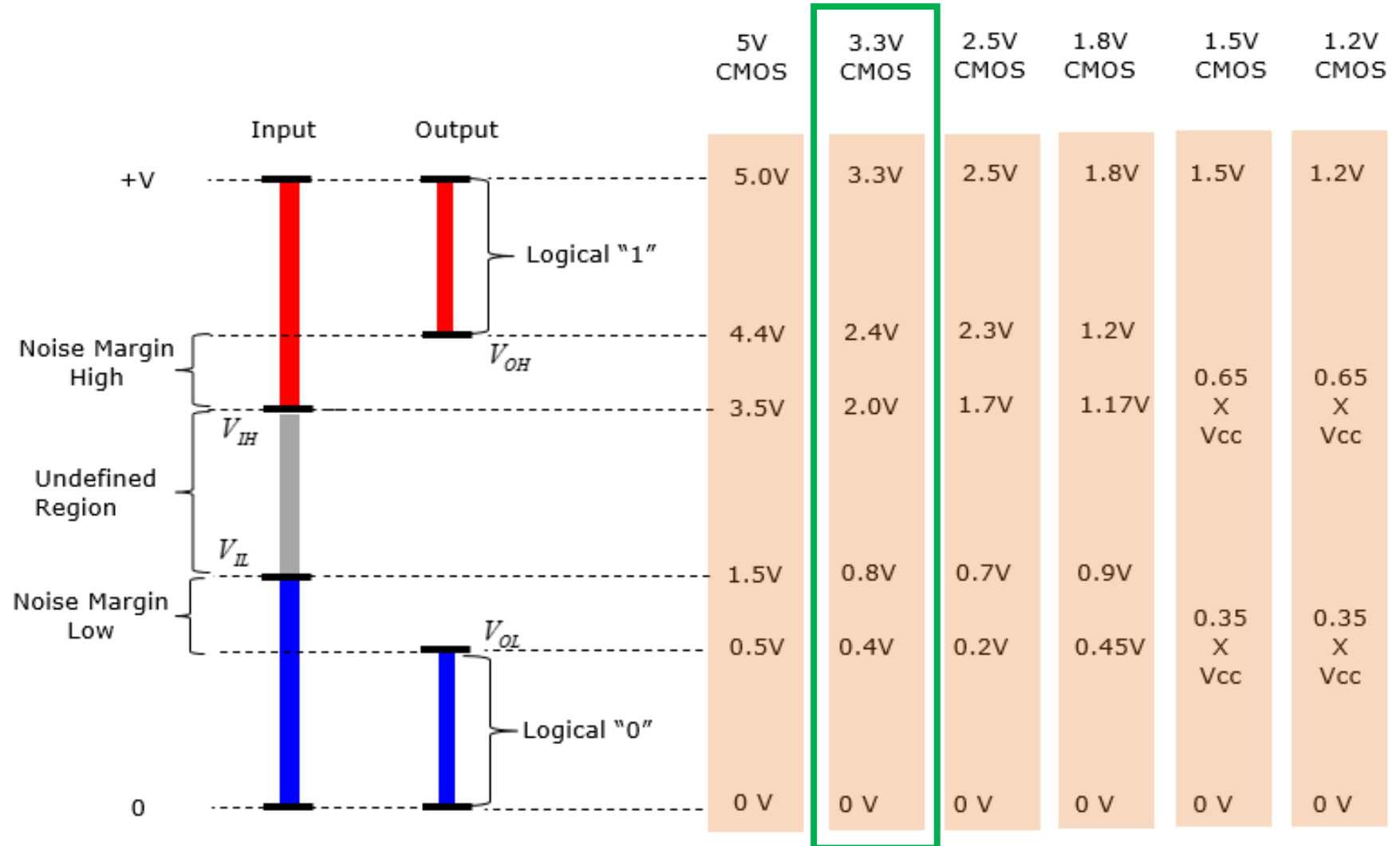
# Digital signals

- Simplest form of I/O
- Exist in two states:
  - High (a.k.a. Set, a.k.a. 1)
  - Low (a.k.a. Clear, a.k.a. 0)
- Simpler to interact with
  - Constrained to two voltages
  - With quick transitions between the two
- No math for voltage level
  - Either high or low



# Digital signals map to voltage ranges

- Upper range is high signal
  - $\sim 0.7 * V_{DD}$
- Bottom range is low signal
  - $\sim 0.3 * V_{DD}$
- Middle is undefined
  - Only exists during transitions

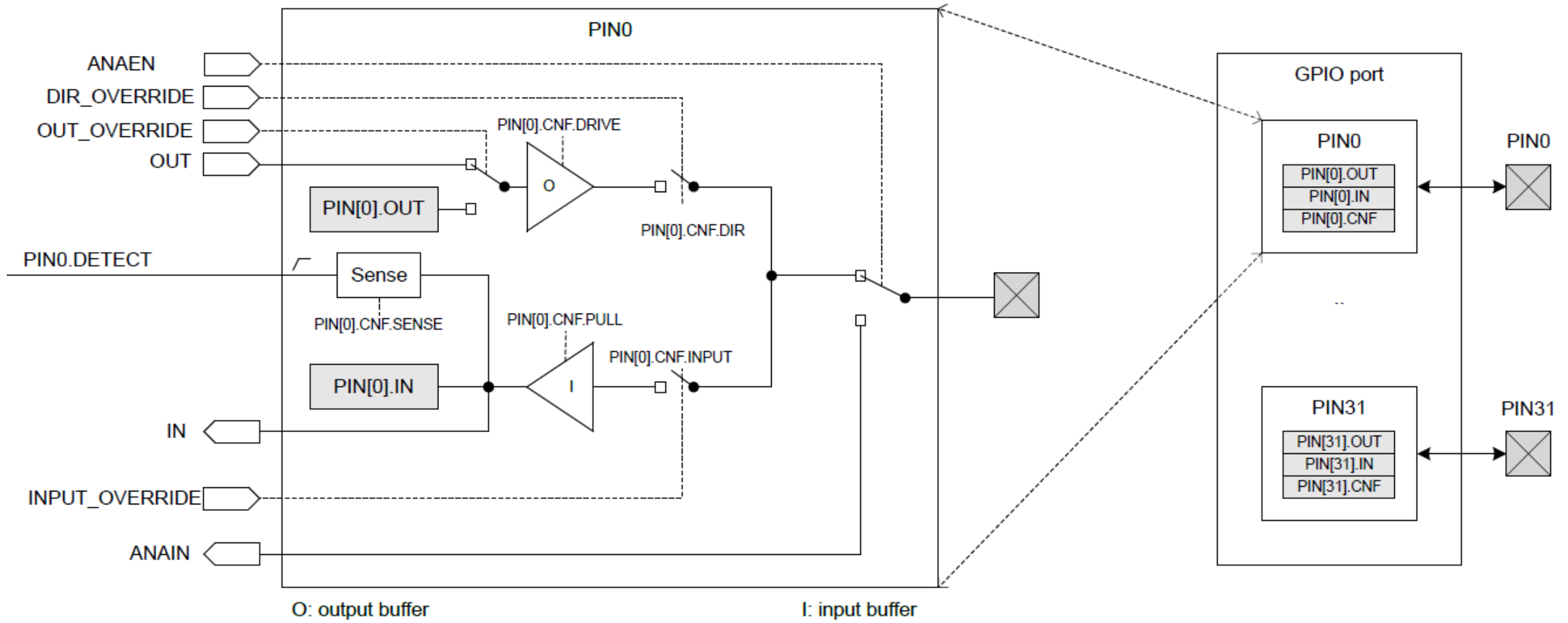


[http://www.sharetechnote.com/html/Electronics\\_CMOS.html](http://www.sharetechnote.com/html/Electronics_CMOS.html)

# General Purpose Input/Output (GPIO)

- Read/write from/to external pins on the microcontroller
  - Two possible values: high (1) or low (0)
- Basic unit of operation for microcontrollers
  - Allows them to interact with buttons and LEDs
  - Every microcontroller has GPIO

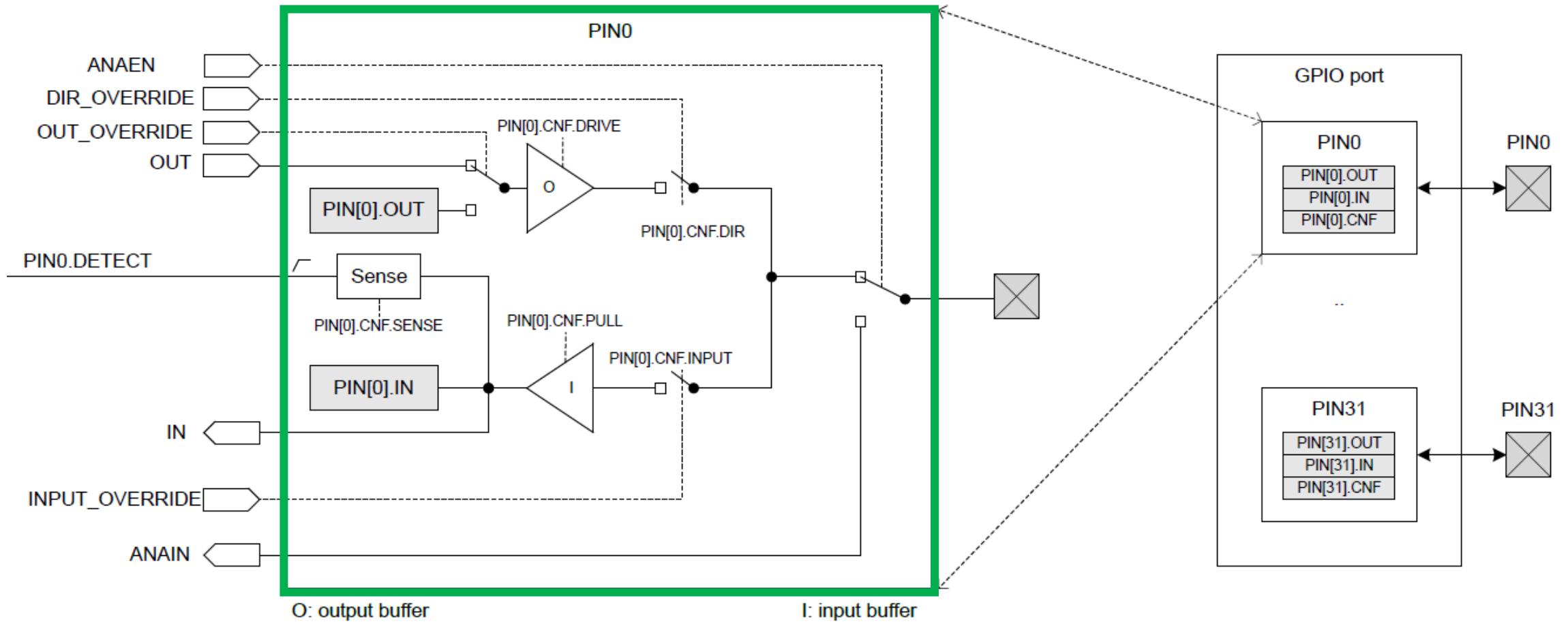
# GPIO on nRF52833



# GPIO on nRF52833

Abstract model of the pin.

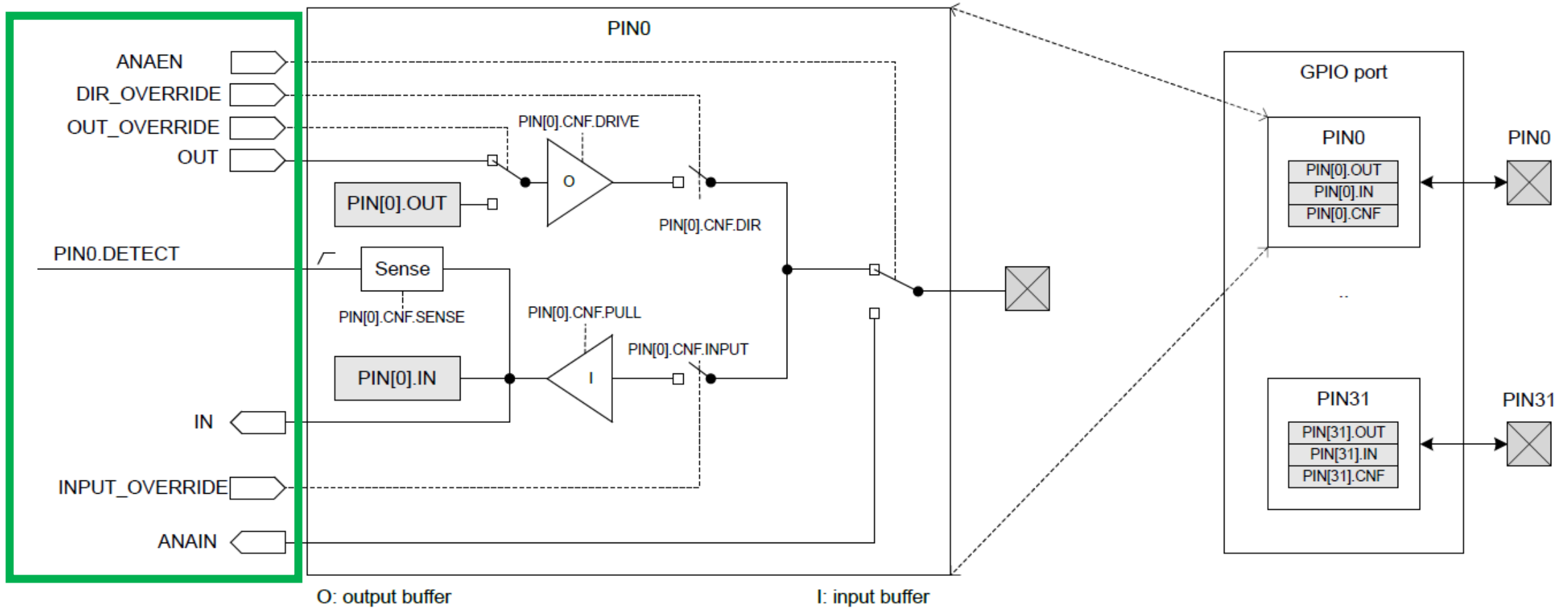
This isn't really how the hardware is implemented. But it's a reasonable model for users.



# GPIO on nRF52833

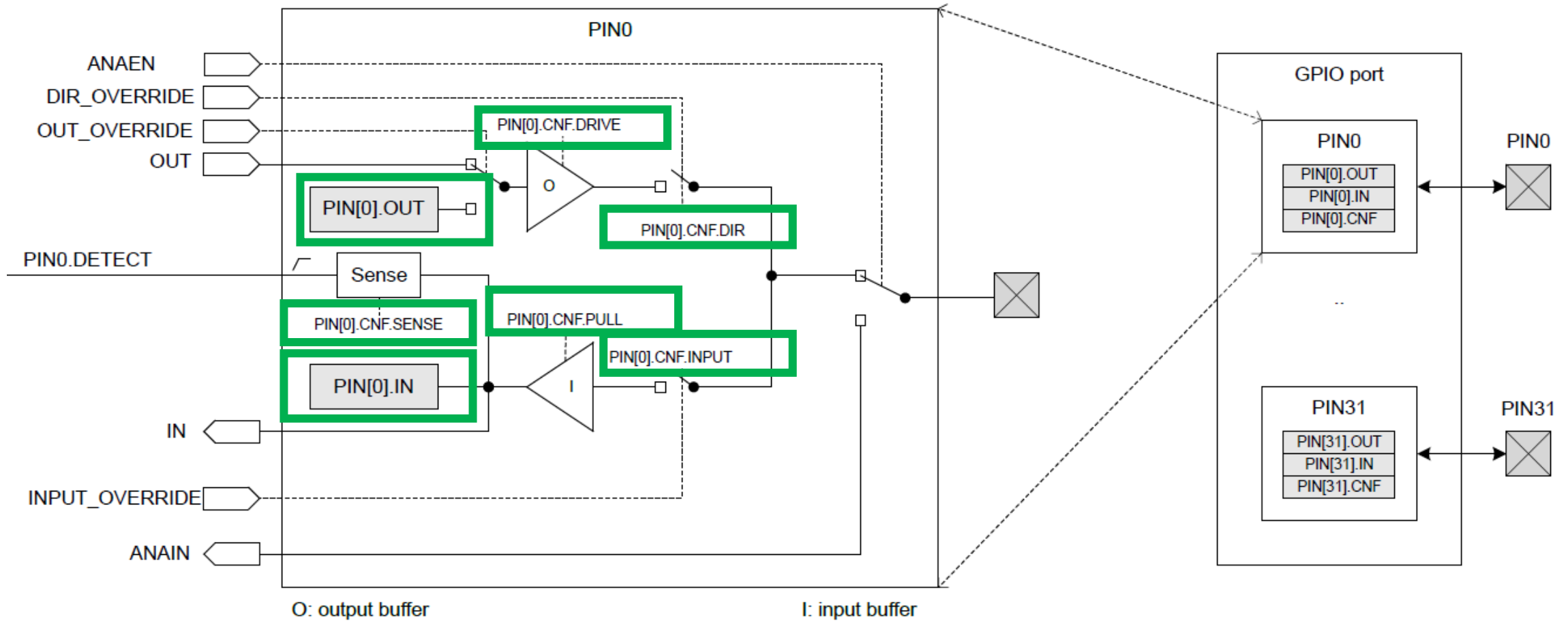
Inputs and outputs to/from the peripheral.

GPIO could be controlled by other peripherals. Controlling a pin in use by other peripherals is bad.



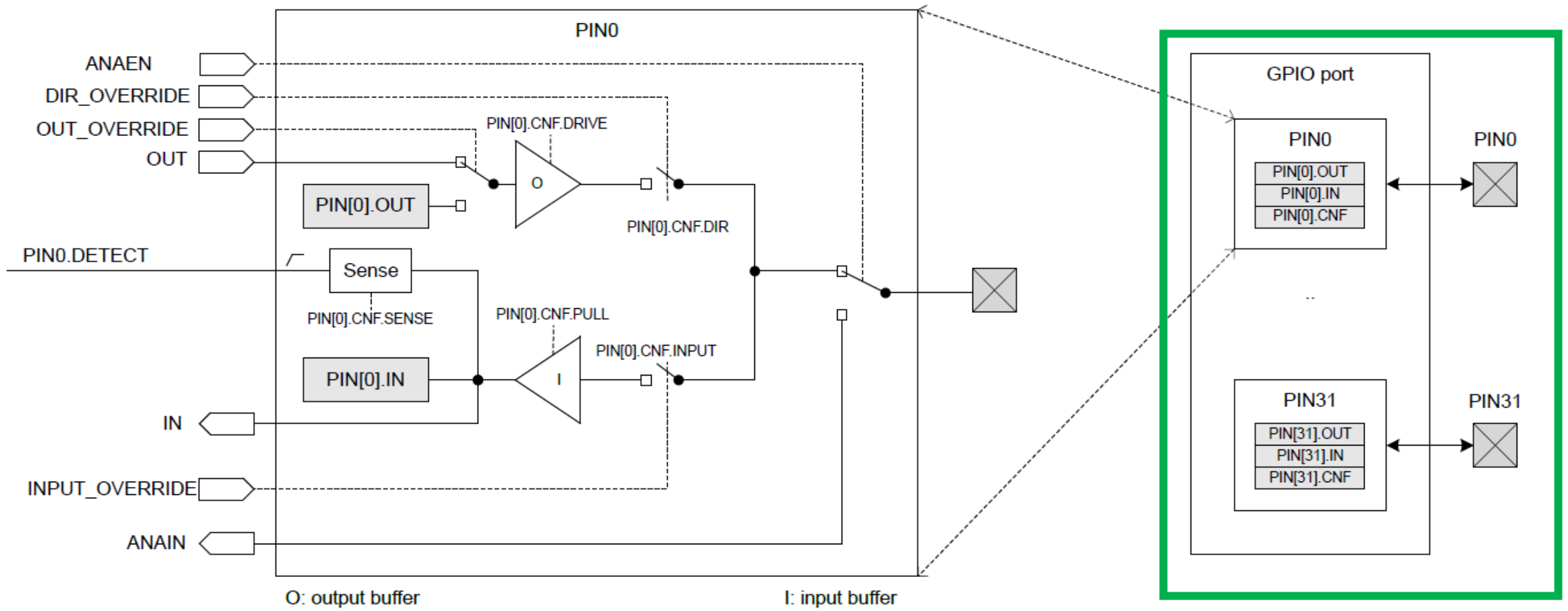
# GPIO on nRF52833

Registers within the GPIO peripheral.  
Configure various things about setup.



# GPIO on nRF52833

Peripheral contents are duplicated for each output pin.  
Each pin has its own registers (or portions thereof).





# Multiple ports

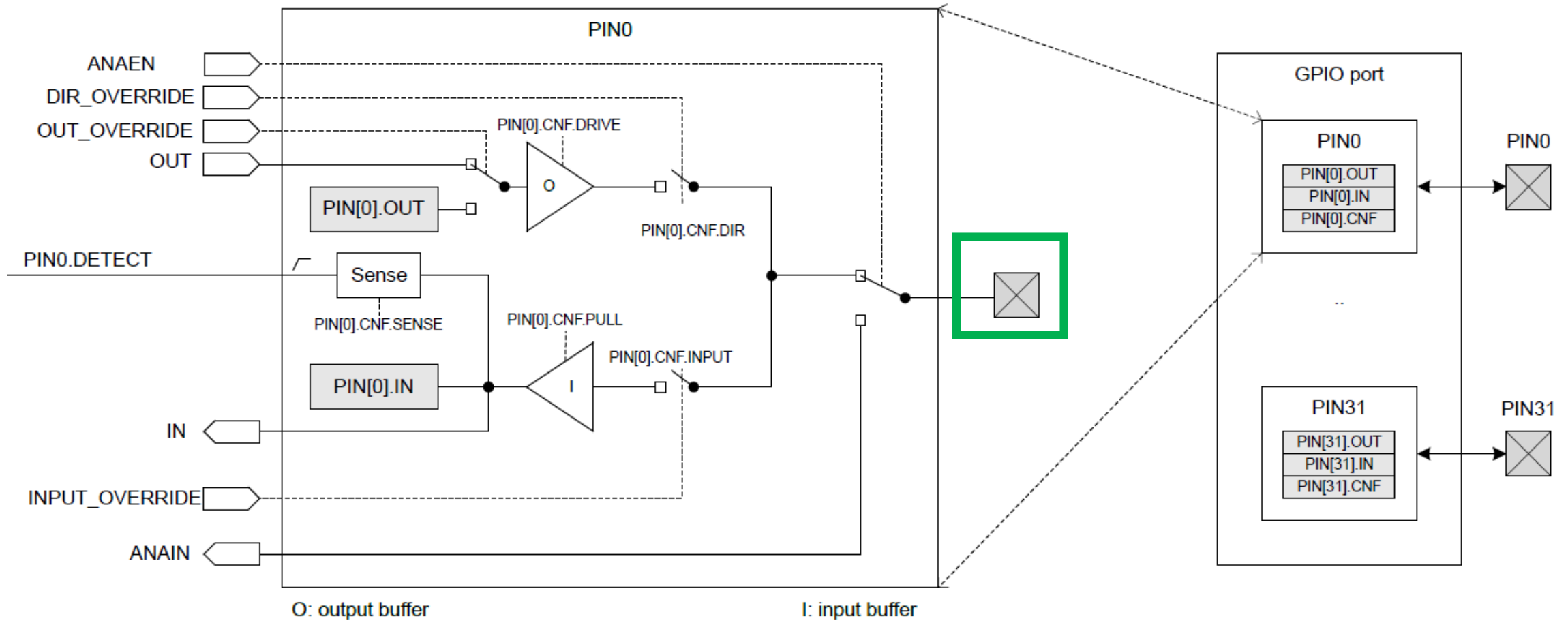
- nRF52833 has up to 42 I/O pins
  - But only 32 can fit in a single word
  - Splits them into two “ports”

Base address	Peripheral	Instance	Description	Configuration
0x50000000	GPIO	GPIO	General purpose input and output	Deprecated
0x50000000	GPIO	P0	General purpose input and output, port 0	P0.00 to P0.31 implemented
0x50000300	GPIO	P1	General purpose input and output, port 1	P1.00 to P1.09 implemented

- Pins are named based on port
  - P0.14 – Button A, P0.23 – Button B
  - P1.04 – LED column 4

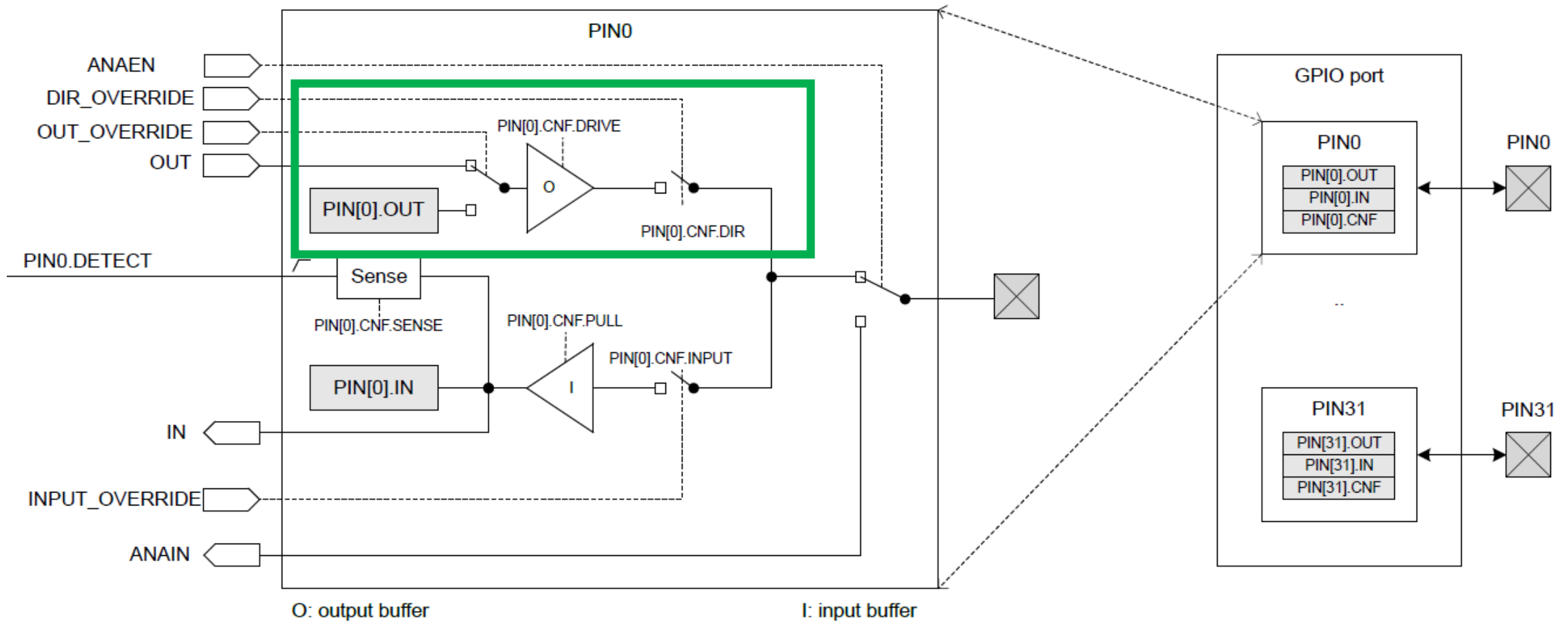
# GPIO on nRF52833

External pin on the microcontroller



# GPIO on nRF52833

Output chain. Signal comes from OUT register, through output buffer, to external pin.

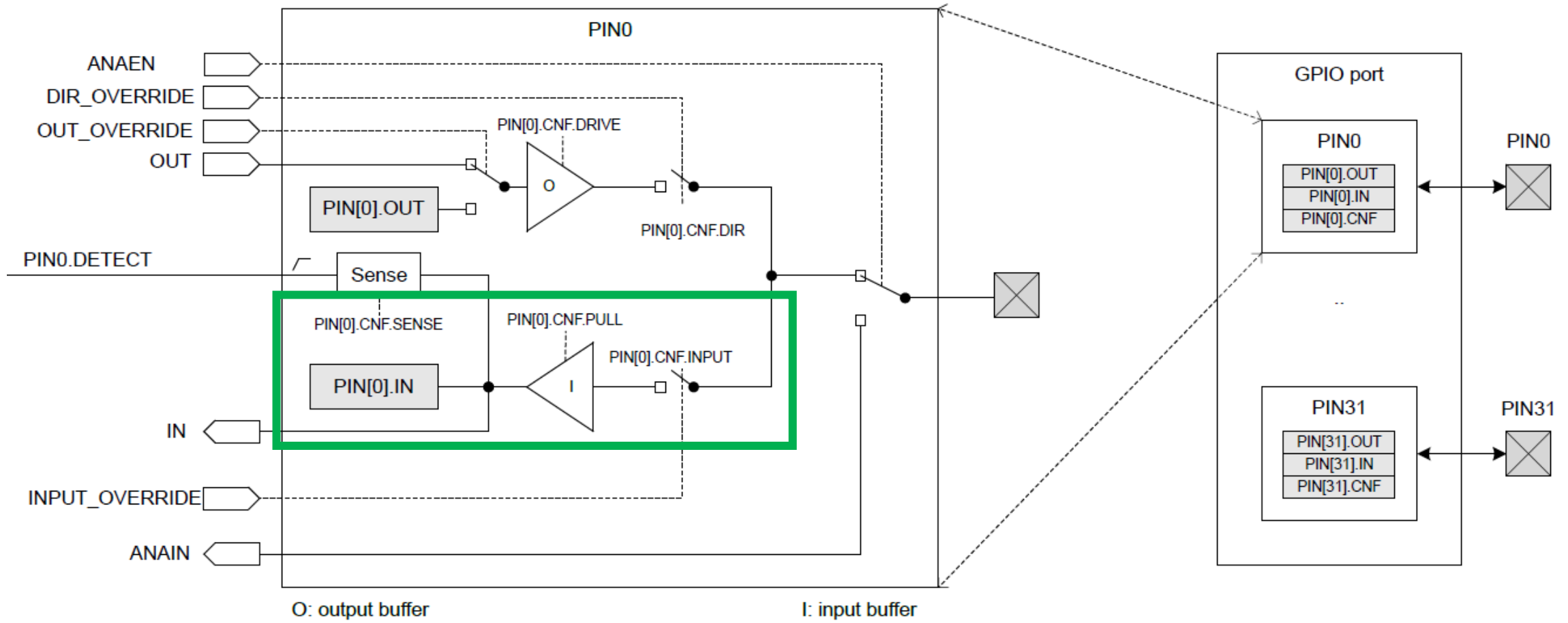


# GPIO Output

- Outputs a high or low signal
- Output configurations
  - High drive output (either for high, low, or both)
    - Sources or sinks additional current
      - For powering external devices
    - Normal drive:  $\sim 2$  mA
    - High drive:  $\sim 10$  mA
  - Disconnect (a.k.a. High Impedance or High-Z)
    - Wired-OR or Wired-AND scenarios

# GPIO on nRF52833

Input chain. Signal goes from pin, through input buffer, to IN register.



# GPIO Input

- Reads in a signal as either high or low
- Input Configurations
  - Input buffer connect/disconnect
    - Allows the pin to be disabled if not being read from
  - Pull
    - Disabled, Pulldown, Pullup
    - Connects an internal pull up/down resistor ( $\sim 13 \text{ k}\Omega$ )
    - Sets default value of input

# Electrical specifications

- High voltage range:  $0.7 \cdot V_{DD}$  to  $V_{DD}$  ( $\sim 2.3$  volts)
- Low voltage range: Ground to  $0.3 \cdot V_{DD}$  ( $\sim 1$  volt)
  
- GPIO are extremely fast
  - Transition time is  $< 25$  ns
  - Connected directly to memory bus for faster interactions
  
- This allows complicated signal patterns to be replicated in software
  - If they aren't implemented as a hardware peripheral
  - Known as "bit-banging"

# Set/Clear registers

Register	Offset	Description
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register

- OUT works traditionally: write a 1 for high, 0 for low
- OUTSET write a 1 to set that pin (high) zero has no effect
- OUTCLR write a 1 to clear that pin (low) zero has no effect
  - Lets you modify a pin without modifying the others (or reading first)



# Outline

- I/O Motivation
- Memory-Mapped I/O
- Interrupts
- **Controlling digital signals**
  - GPIO
  - **GPIOTE**
- DMA

# Handling interrupts from GPIO

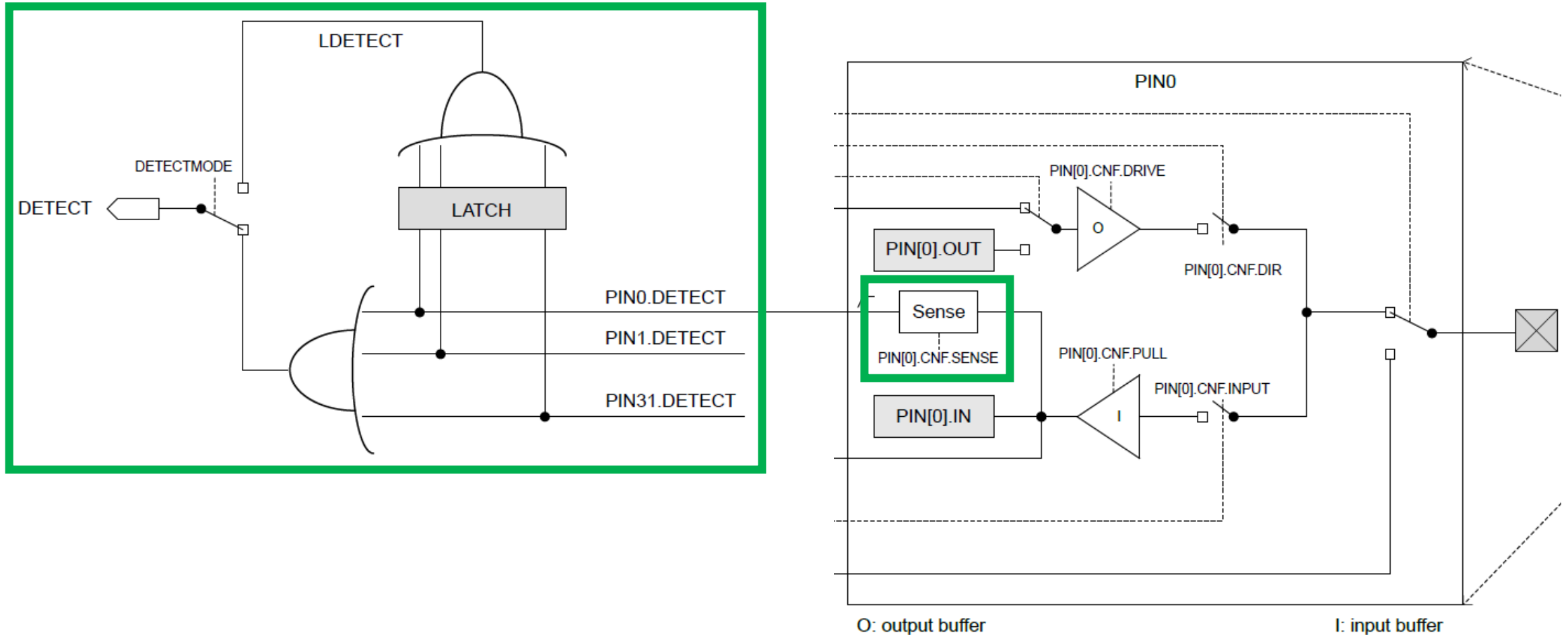
- Separate peripheral, GPIOTE (GPIO Task/Event)
  - Manages up to 8 individual pins
    - Can read inputs and trigger interrupts
    - Can also connect outputs from events on other peripherals (PPI)
  - Can trigger interrupts for a “Port event” as well
    - Software checks which pin(s) caused the event to occur
    - Very low power operation (works with system clocks off)
- Unclear why this is a separate peripheral
  - Presumably too complicated/expensive to have 42 of them

# Configuring individual input interrupts

- Pick an available GPIOTE channel (0-7)
- Configure it
  - Port and Pin number
  - Task (output), Event (input), or Disabled
  - Polarity for input events
    - Low-to-high
    - High-to-low
    - Toggle (both directions)
- Enable interrupts for channel in GPIOTE (and in NVIC!)
- Clear event in interrupt handler
  - Doesn't happen automatically

# Sensing port events

- Uses the "Detect" signal. Generated from pin Sense configuration



# Configuring port input interrupts

- Configure the Sense for each pin
  - High or Low
  - Allows different pins to have different “active” states
- Select detect mode
  - Direct connection to pins
  - Latched version (saved even if pin later changes back)
- Enable interrupts for port in GPIOTE (and in NVIC!)
- Clear event in interrupt handler and value in Latch register
  - Doesn't happen automatically

# Break

# Outline

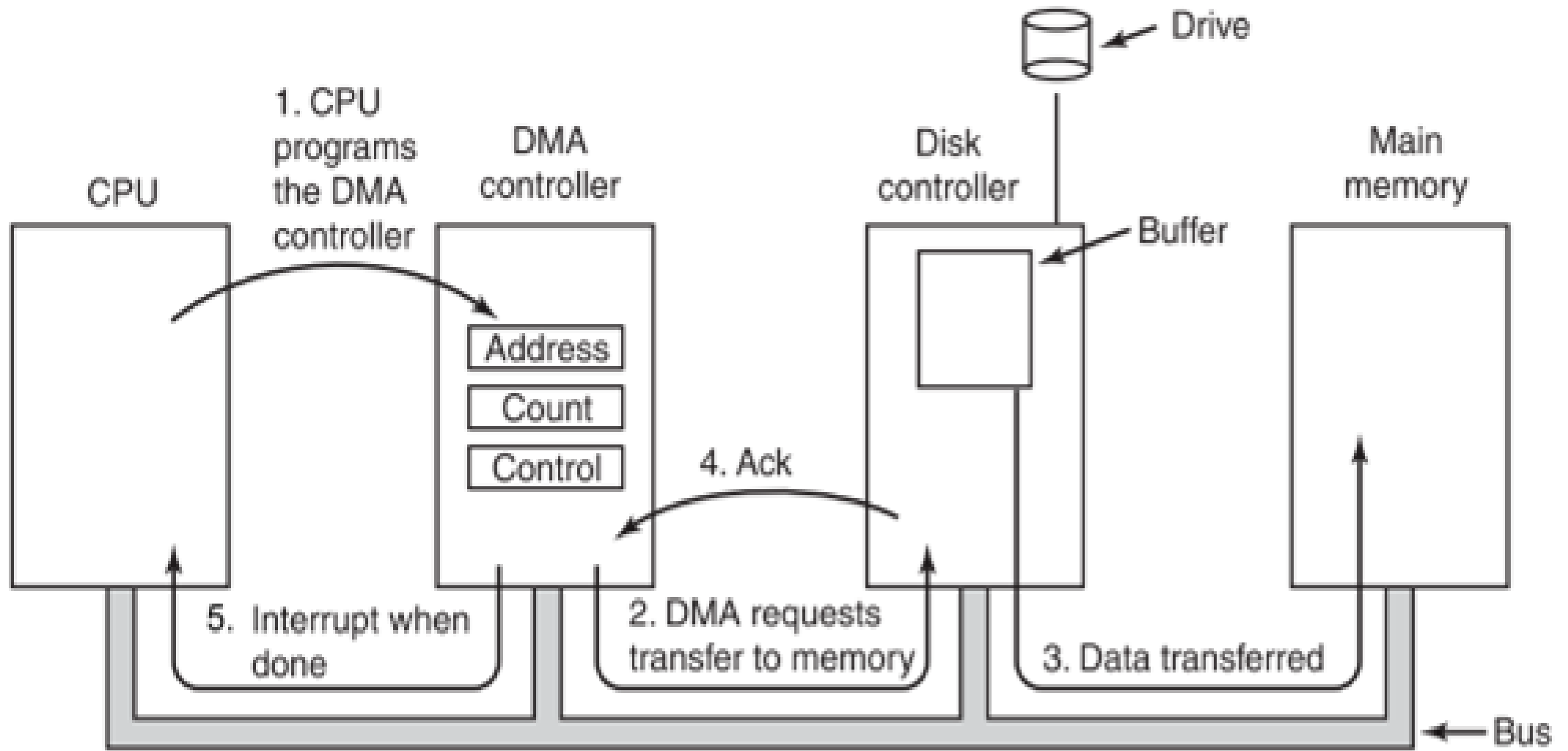
- I/O Motivation
- Memory-Mapped I/O
- Interrupts
- Controlling digital signals
  - GPIO
  - GPIOTE
- **DMA**

# Direct Memory Access (DMA)

- Even with interrupts, providing data to the peripheral is time consuming
  - Need to be interrupted every byte, to copy the next byte over
- DMA is an alternative method that uses hardware to do the memory transfers for the processor
  - Software writes address of the data and the size to the peripheral
  - Peripheral reads data directly from memory
  - Processor can go do other things while read/write is occurring

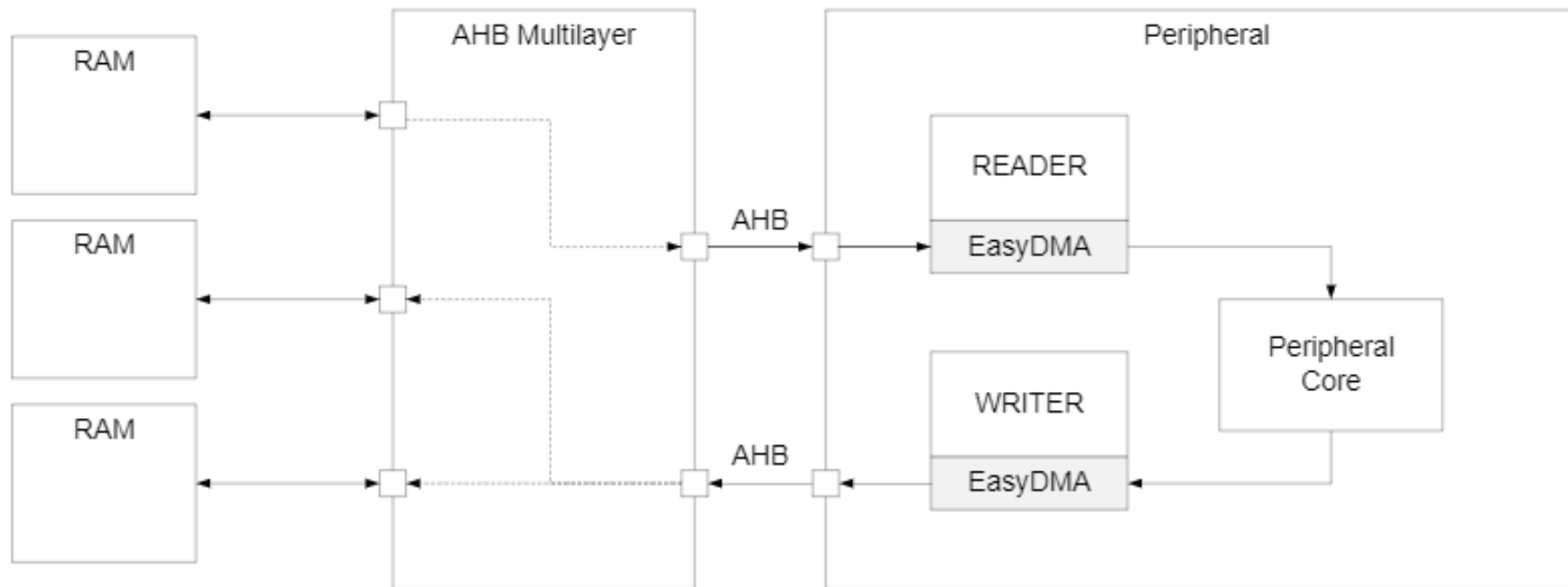


# General-purpose DMA



# Special-purpose DMA

- nRF52 uses “EasyDMA”, which is built into individual peripherals
  - Only capable of transferring data in/out of that peripheral
  - Easier to set up and use in practice
  - Only available on some peripherals though (no DMA for TEMP)



**Warning:** addresses for DMA  
MUST be in RAM!

# Full peripheral interaction pattern

1. Configure the peripheral
2. Enable peripheral interrupts
3. Set up peripheral DMA transfer
4. Start peripheral

Continue on to other code

5. Interrupt occurs, signaling DMA transfer complete
6. Set up next DMA transfer

Continue on to other code, and repeat

# Outline

- I/O Motivation
- Memory-Mapped I/O
- Interrupts
- Controlling digital signals
  - GPIO
  - GPIOTE
- DMA