Lecture 04
Input and Output

CE346 – Microprocessor System Design

Branden Ghena – Fall 2023

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

Northwestern

Administrivia

- Lab on Friday!
  - Frances Searle room 3220
  - See you all there!!
  - Show up on-time. We are going to get started right away
    - None of this 5-10 minutes late nonsense

- Bring a USB-C adapter if you need one

- Remember that you need to attend the section you registered for in CAESAR
  - If you need to switch for a day, ask me for permission in advance on Piazza

Quiz coming soon

- First quiz is next week Tuesday! (10/08)
  - 15-minute quiz, taken in-class on paper
  - Last fifteen minutes of class
  - Bring a pencil
  - No notes, no calculator

- Covers material from the last two weeks, including today

- Goals:
  - The quiz is not meant to be difficult. It’s meant to keep you involved
  - Do review class material and make sure you actually understand it

Today's Goals

- How does a microcontroller interact with peripherals to perform input and output operations?

- Explore reliable use of Memory-Mapped I/O

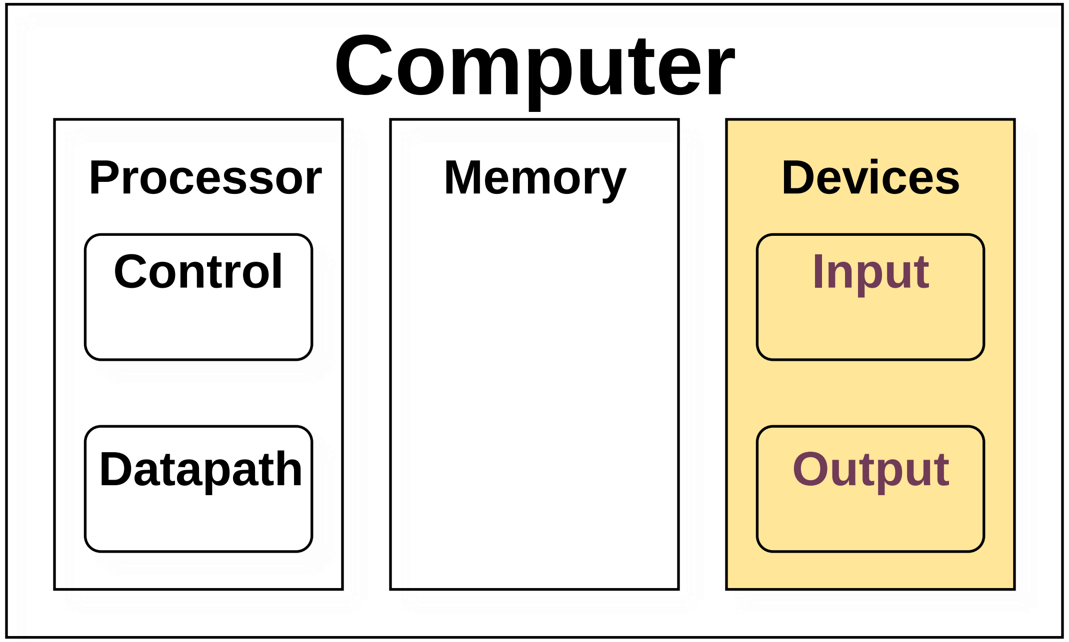
- Learn about our first peripherals: Temperature and GPIO
- Explore General Purpose I/O (GPIO) peripheral use
  - Understand how it works
  - Understand what kinds of configurations it might have

Outline

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

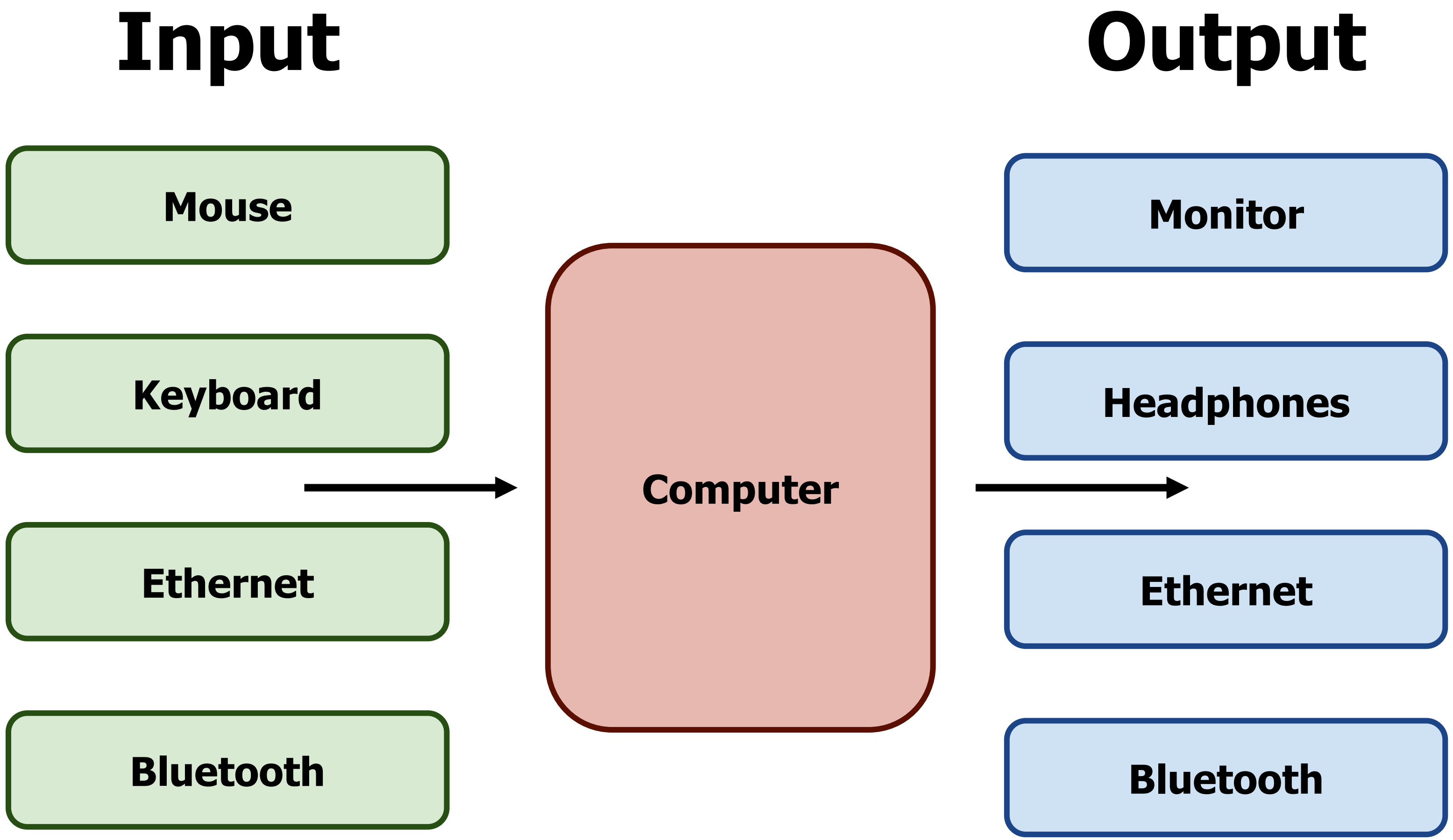
Devices are the point of computers

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

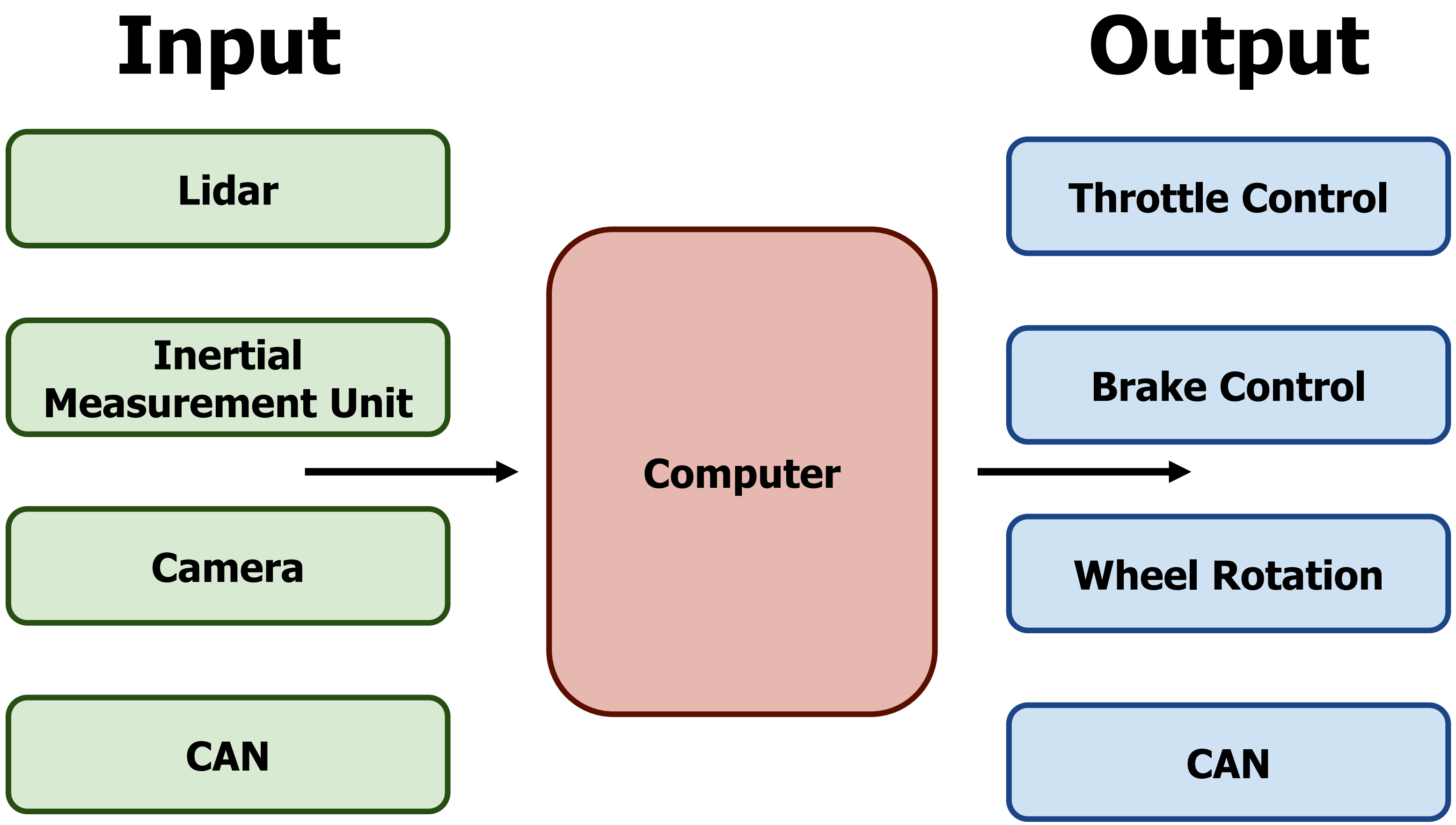


- Embedded systems have the same requirement, just more types of IO

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
- System must be able to handle each of these
  - Sometimes needs low overhead
  - Sometimes needs to not wait around

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

Outline

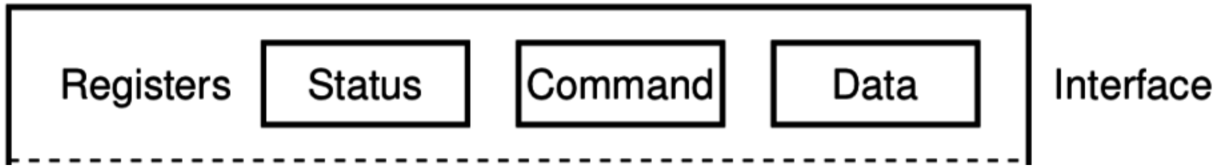
- I/O Motivation
- **Memory-Mapped I/O**
- Controlling digital signals
  - GPIO
  - GPIOTE

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

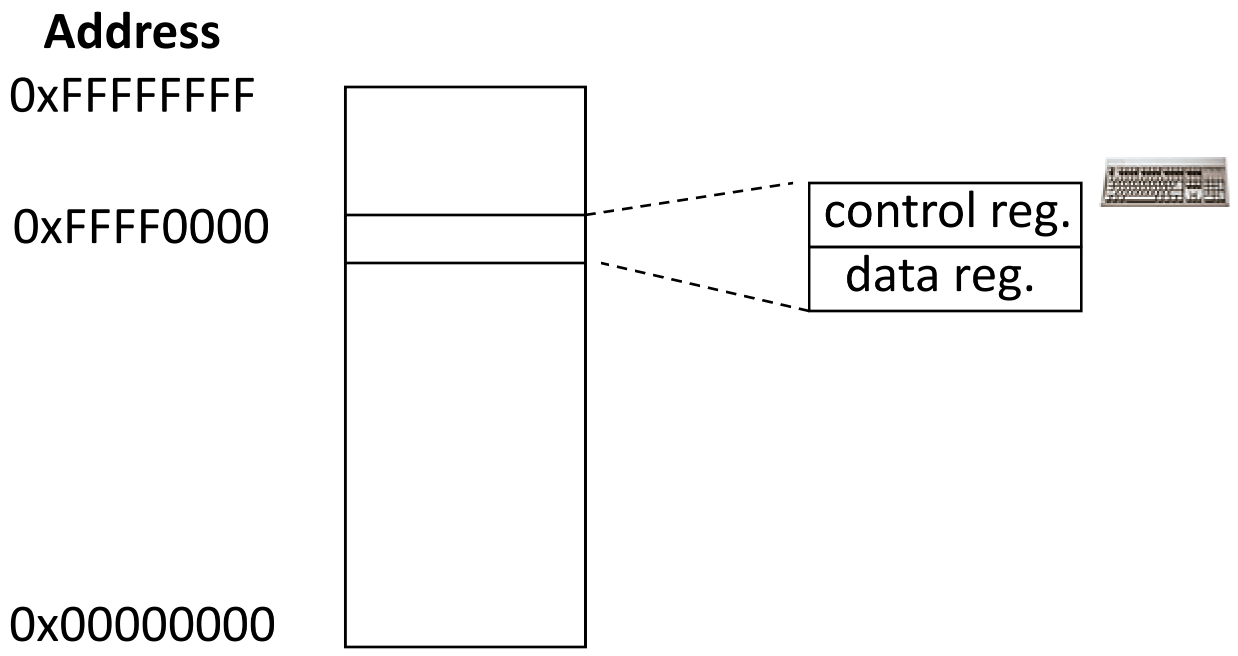
- We interact with a peripheral's interface
  - Called registers (actually are from EE perspective, but you can't use them)
  - Read/Write like they're data

- 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!



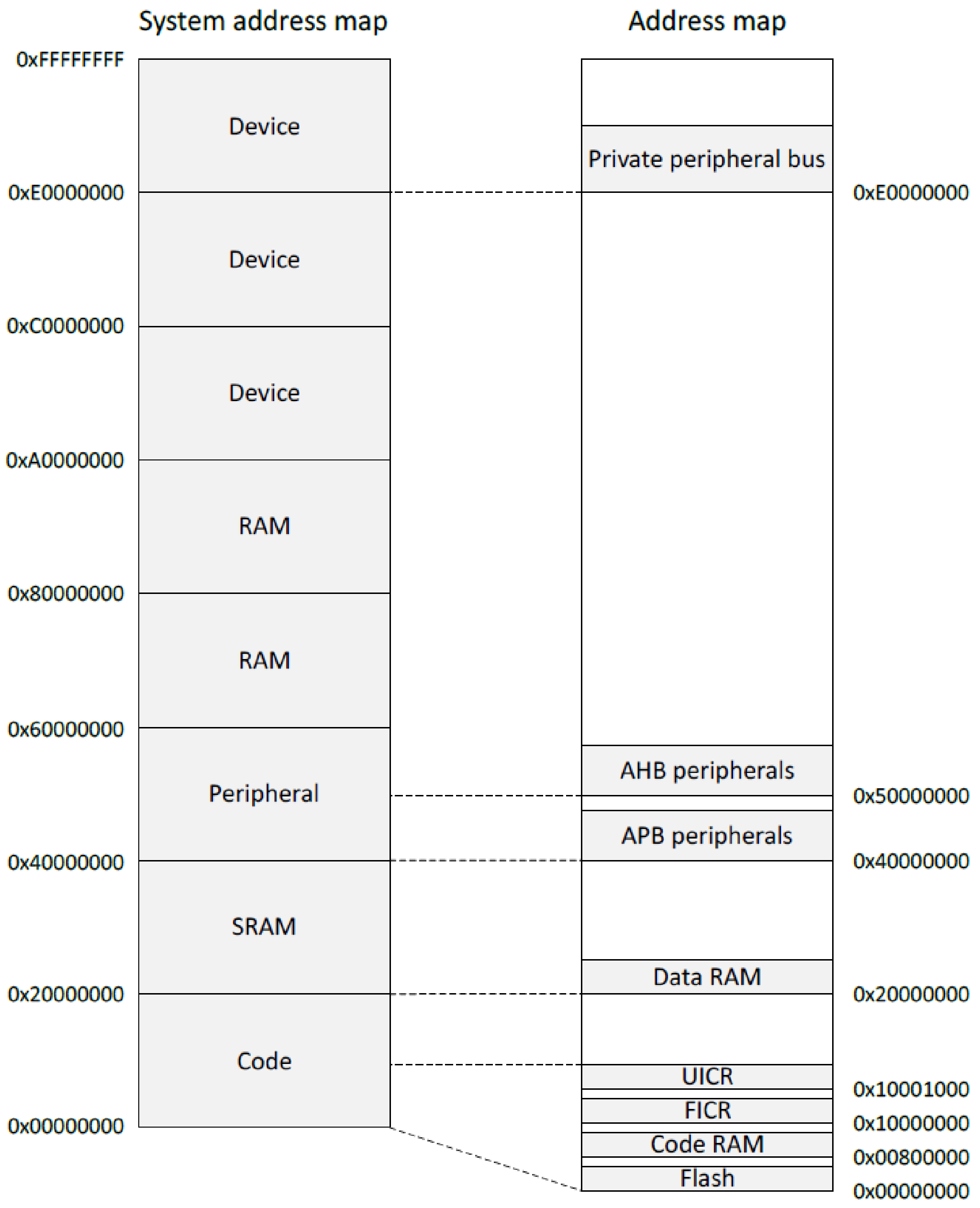
- Every microcontroller I've ever seen uses MMIO

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!

- Let’s write it from scratch

Example code (`temp_mmio` app)

```c
// 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[0x204/4 - 1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[(0x508 - 0x308)/4 - 1];
    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)

With increasingly verbose ways to write the size of the “unused” space (any of these will do, but don’t forget the -1)

Temperature peripheral MMIO struct

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)

```
typedef struct {
    uint32_t TASKS_START;
    uint32_t TASKS_STOP;
    uint32_t _unused_A[62];
    uint32_t EVENTS_DATARDY;
    uint32_t _unused_B[0x204/4 - 1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[(0x508 – 0x308)/4 – 1];
    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;
```

Break + relevant xkcd

https://xkcd.com/138/

Outline

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

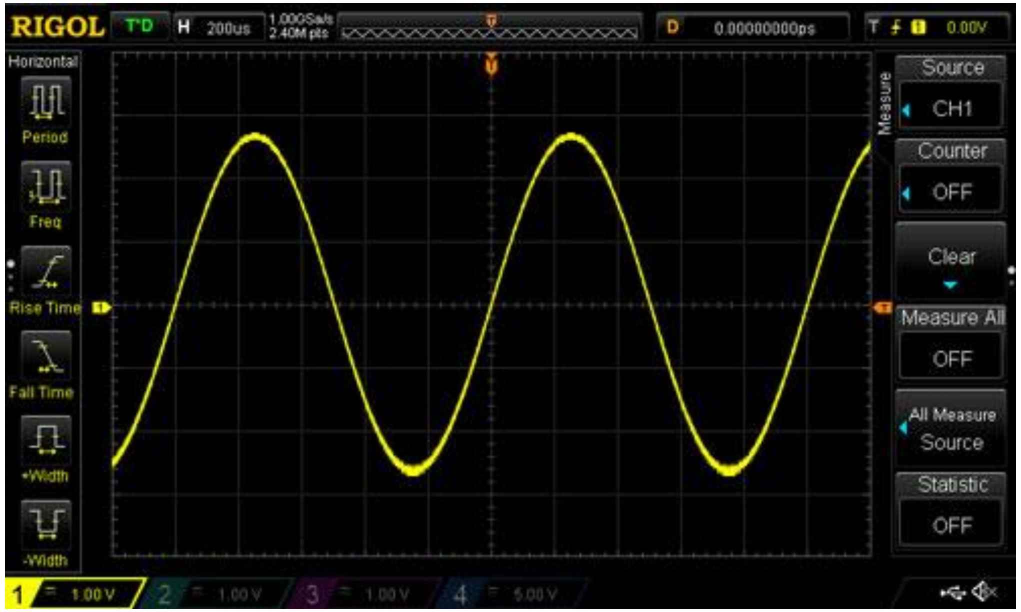
Digital signals

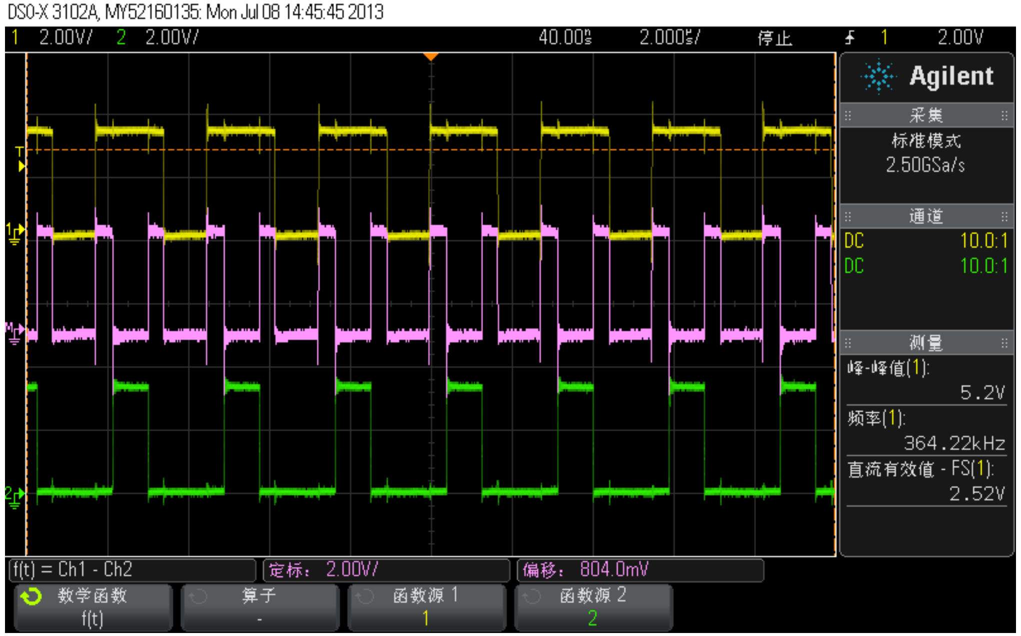
- Simplest form of I/O

- 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

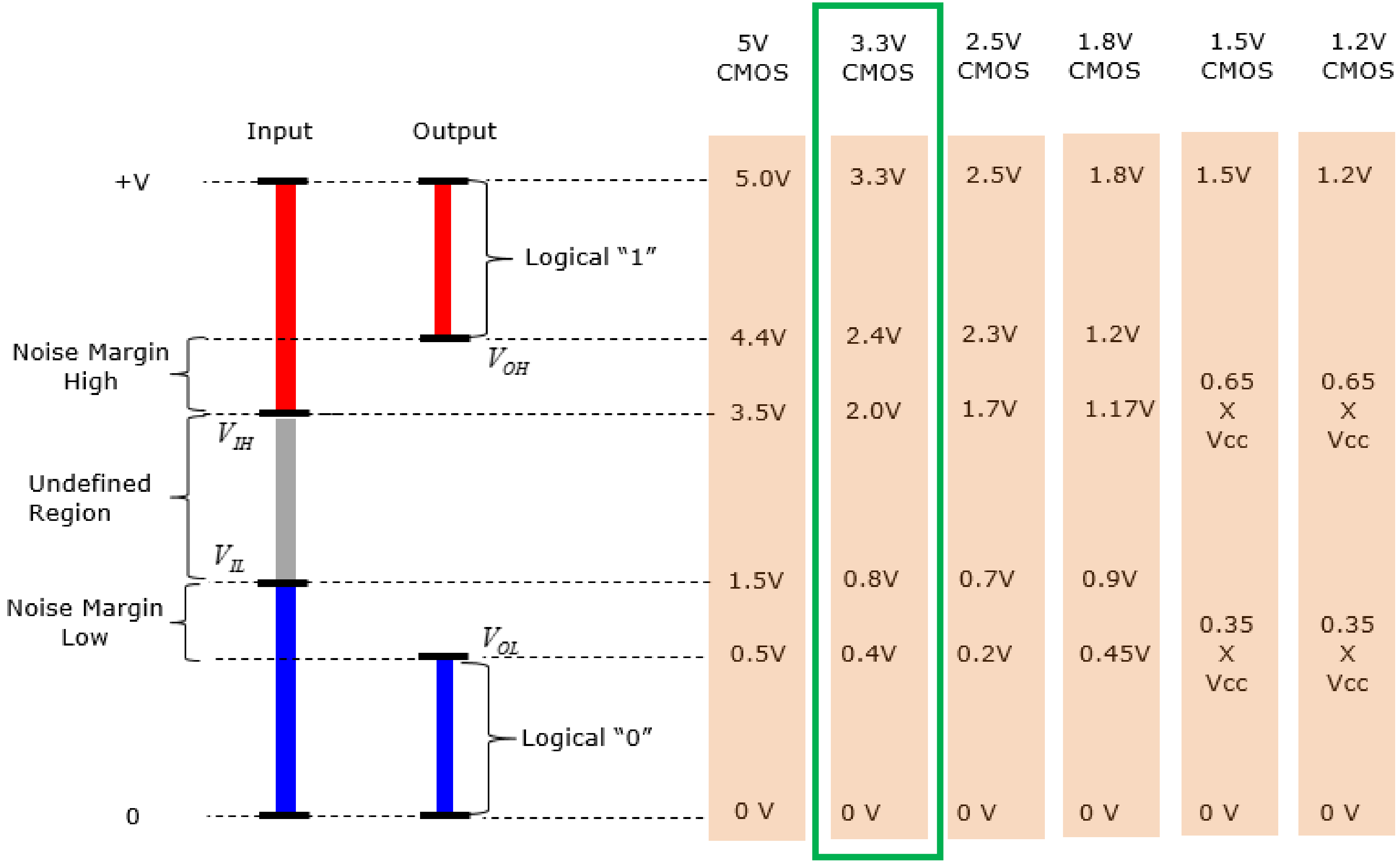
  - No math for voltage level
    - Either high or low





Digital signals map to voltage ranges

- Upper range is high signal
  - ~0.7*VDD
- Bottom range is low signal
  - ~0.3*VDD
- Middle is undefined
  - Only exists during transitions



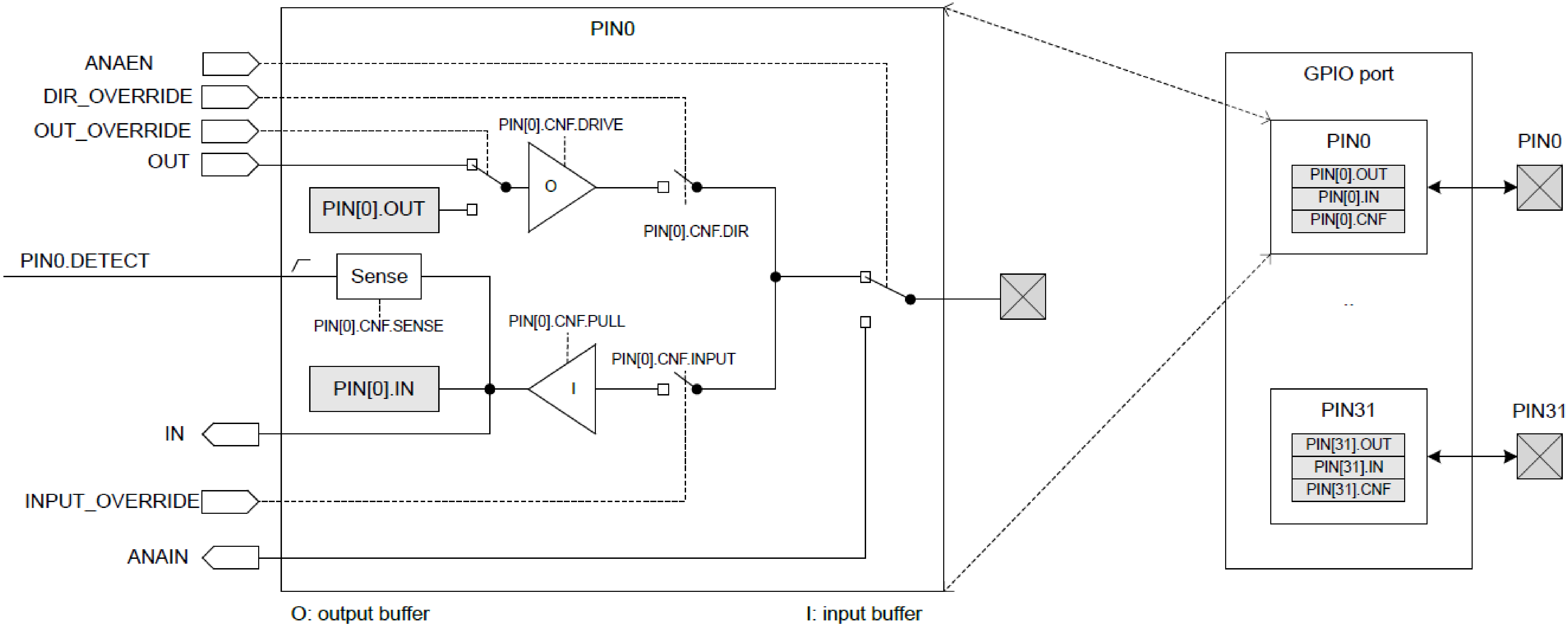
http://www.sharetechnote.com/html/Electronics_CMOS.html

General Purpose Input/Output (GPIO)

- Read/write from/to external pins on the microcontroller
  - Two possible values: high (1) or low (0)

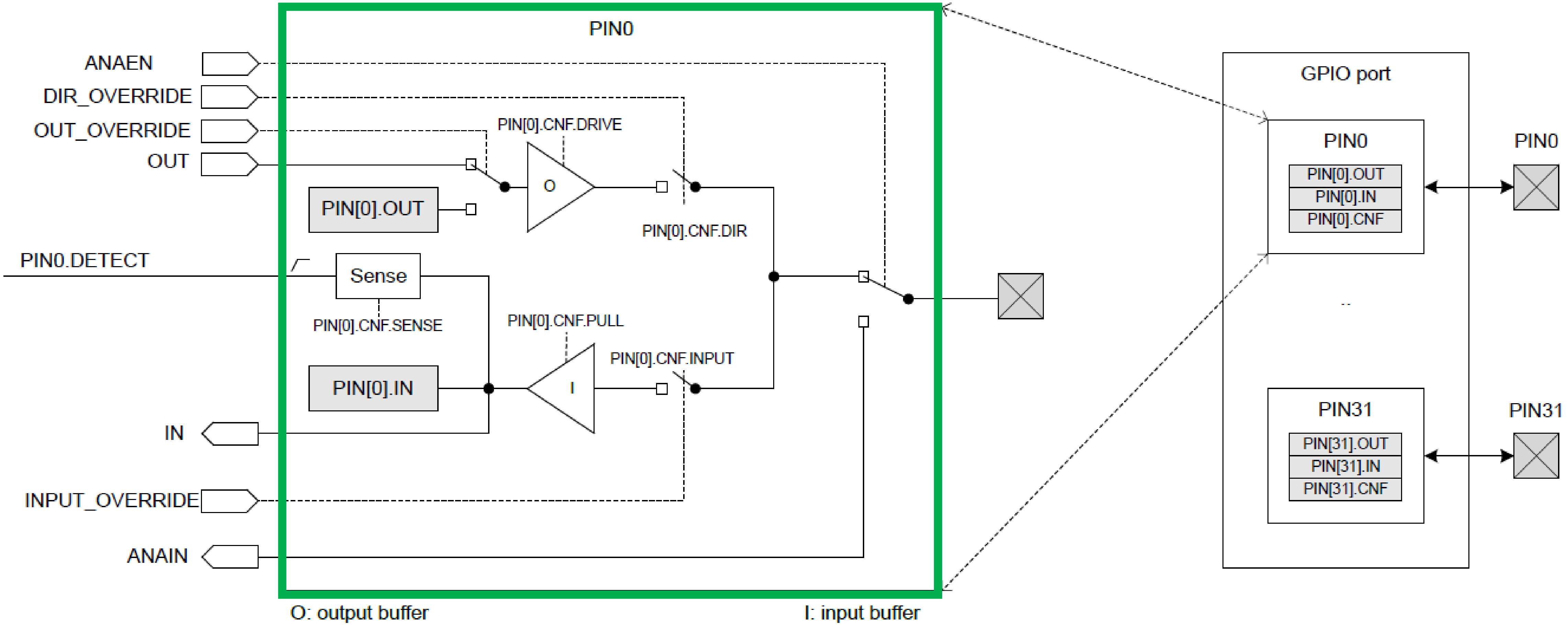
- Basic unit of operation for microcontrollers
  - Allows them to interact with buttons and LEDs
  - Every microcontroller has GPIO

GPIO on nRF52833



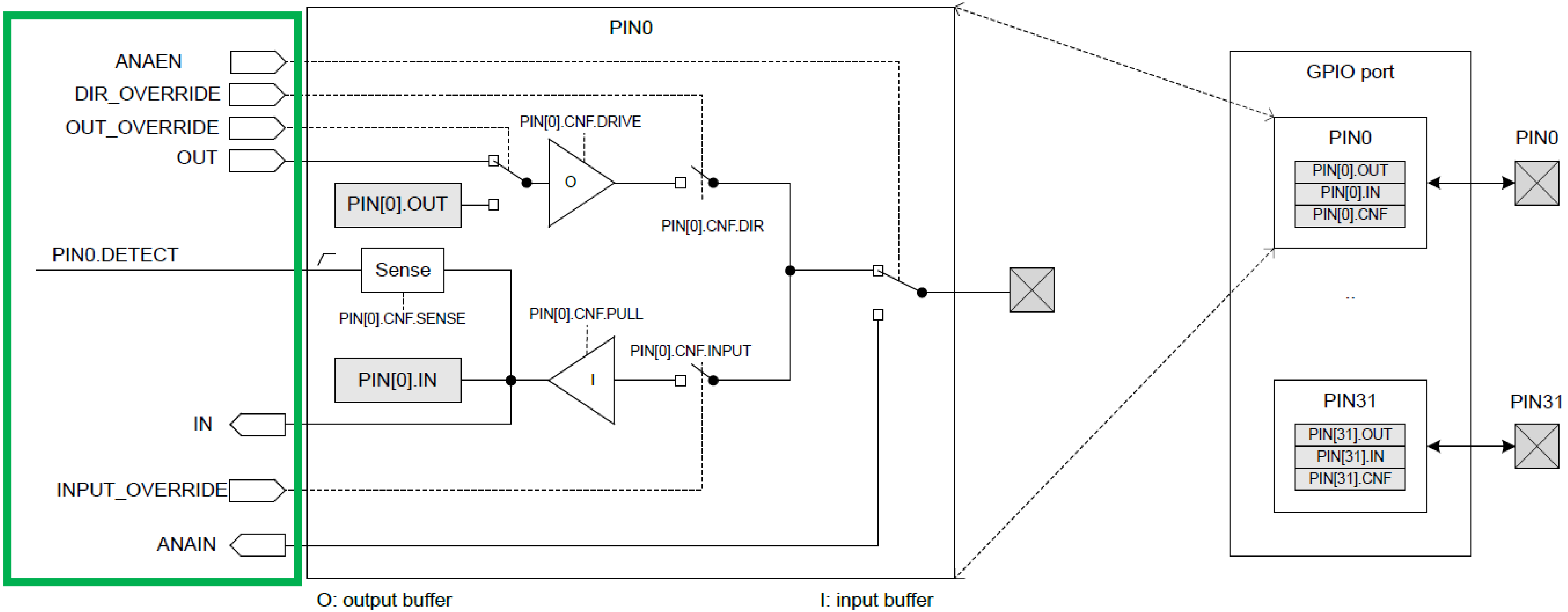
GPIO on nRF52833

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



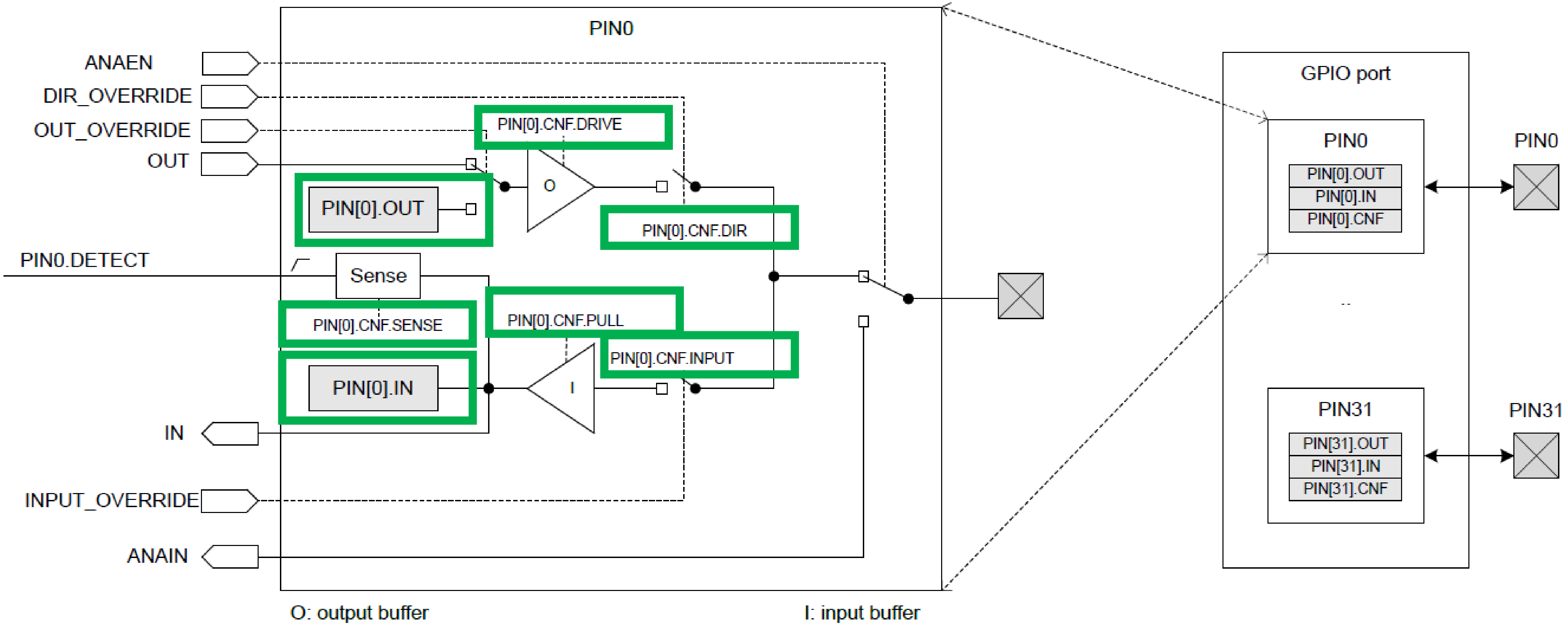
GPIO on nRF52833

Inputs and outputs to/from the peripheral.
GPIO could be controlled by other peripherals. Controlling a pin in use by other peripherals is bad.



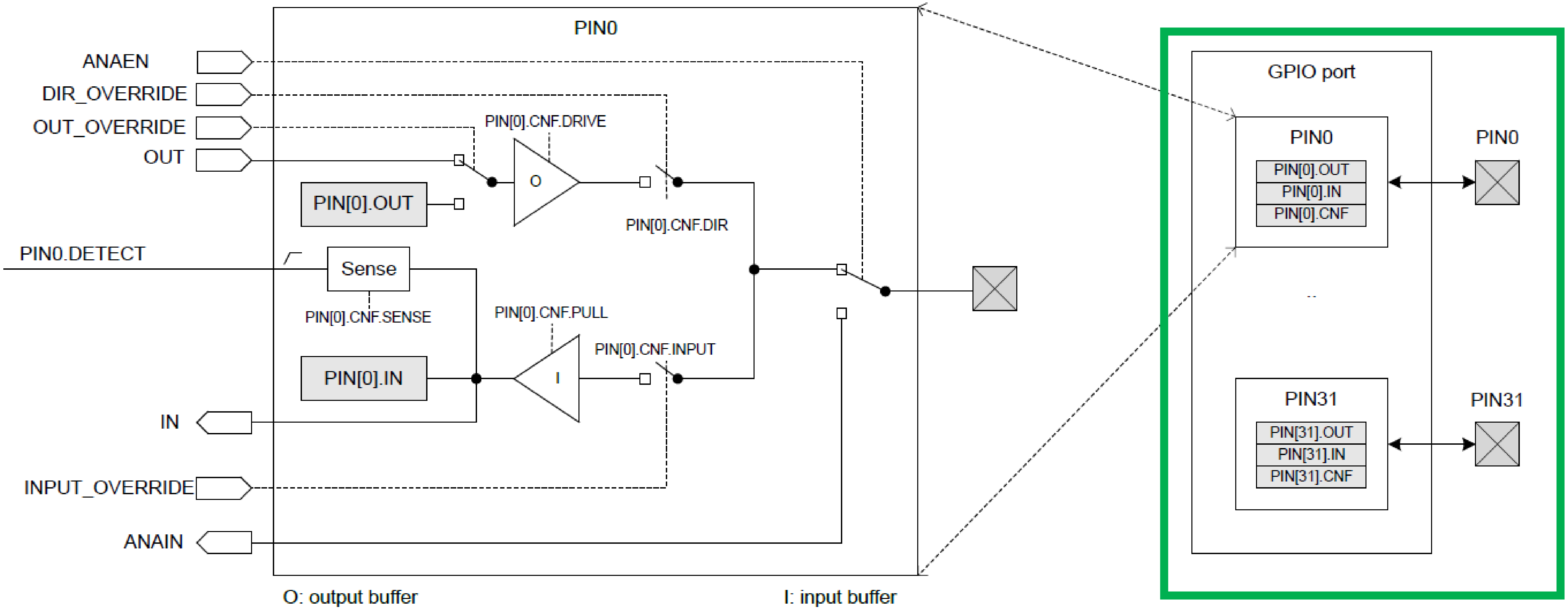
GPIO on nRF52833

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

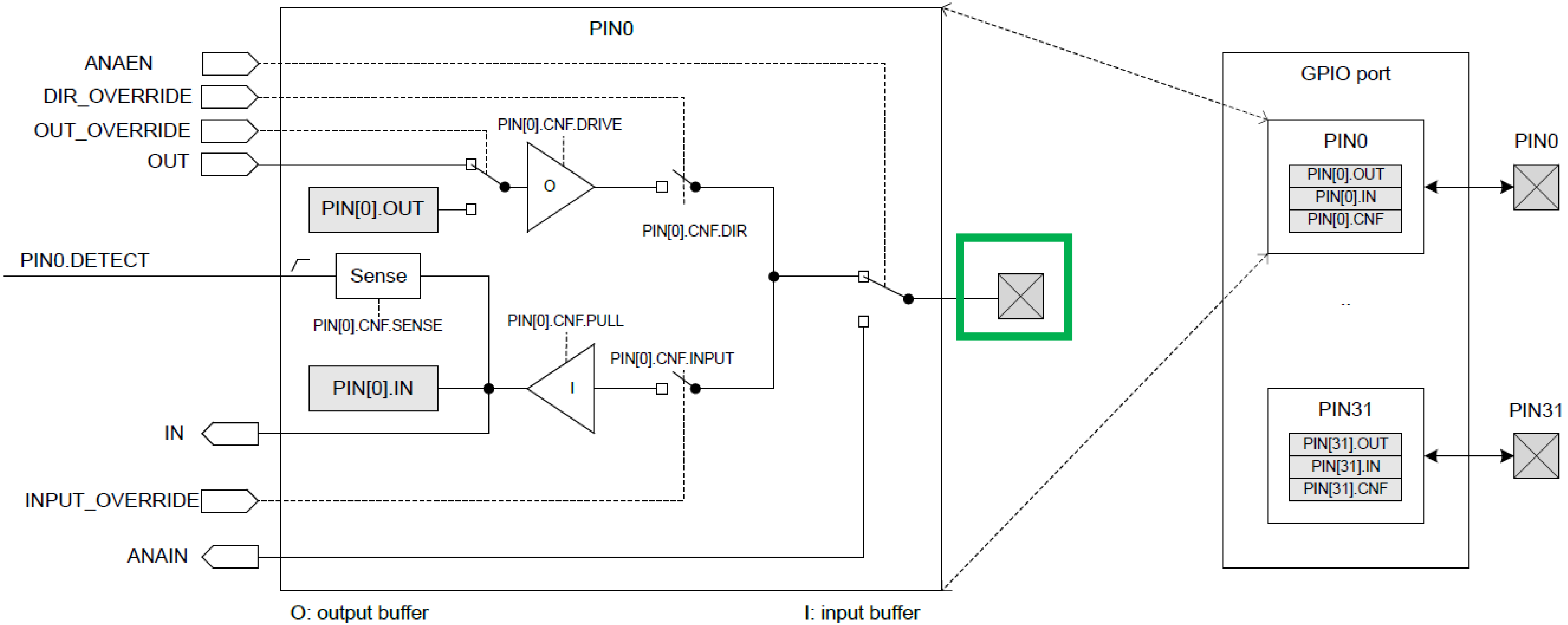
- nRF52833 has up to 42 I/O pins
  - But only 32 can fit in a single 32-bit word
  - Splits them into two “ports”

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

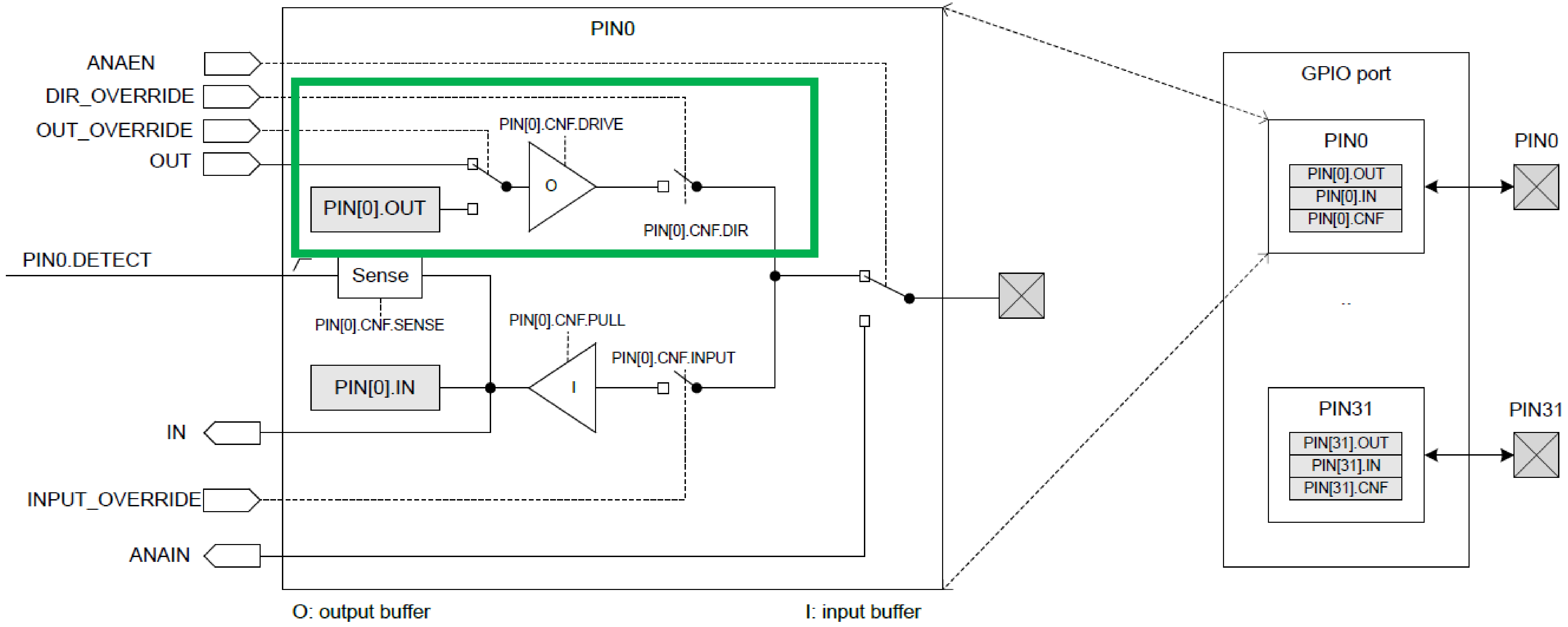
GPIO on nRF52833

External pin on the microcontroller



GPIO on nRF52833

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



GPIO Output

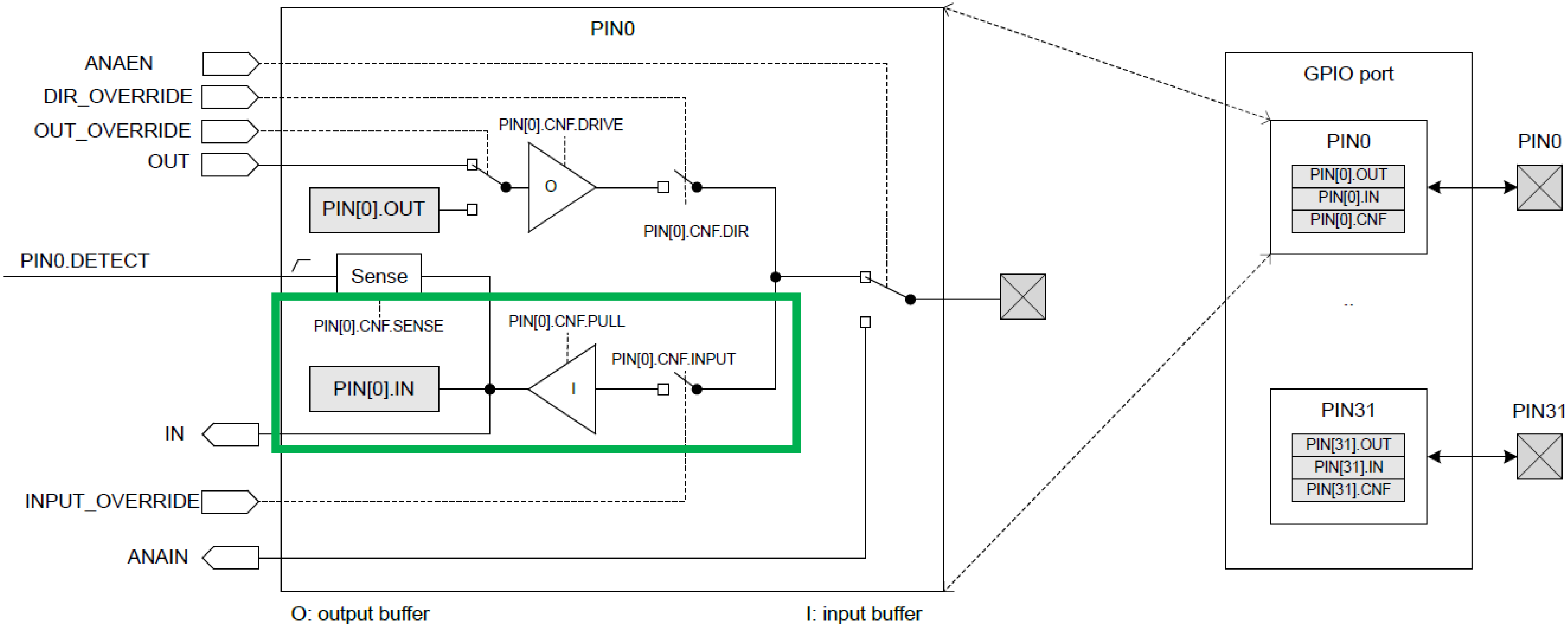
- Outputs a high or low signal

- Output configurations
  - High drive output (either for high, low, or both)
    - Sources or sinks additional current
      - For powering external devices
    - Normal drive: ~2 mA
    - High drive: ~10 mA

  - Disconnect (a.k.a. High Impedance or High-Z)
    - Wired-OR or Wired-AND scenarios (we’ll talk about these later in class)

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 (we'll discuss in a future lecture)
    - Connects an internal pull up/down resistor (~13 kΩ)
    - Sets default value of input

Electrical specifications

- High voltage range: 0.7*VDD to VDD (~2.3 volts)
- Low voltage range: Ground to 0.3*VDD (~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”

Pin configuration

6.8.2.5 DIR

Address offset: 0x514

Direction of GPIO pins

Bit number	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
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**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**

ID	Acce	Field	Value ID	Value	Description
A-f	RW	PIN[i] (i=0..31)			Pin i
			Input	0	Pin set as input
			Output	1	Pin set as output

- DIR register controls direction (input or output) for each pin
  - Each bit 0-31 corresponds to pin 0-31
  - Reset value: 0x00000000 -> all pins are inputs by default

Controlling output level

6.8.2.1 OUT

Address offset: 0x504

Write GPIO port

Bit number	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
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**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**

ID	Acce	Field	Value ID	Value	Description
A-f	RW	PIN[i] (i=0..31)			Pin i
			Low	0	Pin driver is low
			High	1	Pin driver is high

- OUT register controls whether each pin is high or low
  - Only meaningful if the pin is configured as an Output
  - Again, each bit is a single pin and reset is 0x00000000 (all pins low)

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)

Complex configuration

- If you want to change other pin configurations, you do so per pin with the `PIN_CNF[n]` registers
  - There are 32 of them, one per pin
- Various fields correspond to different groups of bits
  - Direction, Input buffer, Pullup/down, Drive strength, Sensing mechanism
- Bits not part of a field should be ignored
  - Do not modify them

Bit number	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
ID															E	E						D	D	D					C	C	B	A
Reset 0x00000002	**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**	**1**	**0**

ID	Acce	Field	Value ID	Value	Description
A	RW	DIR			Pin direction. Same physical register as DIR register
			Input	0	Configure pin as an input pin
			Output	1	Configure pin as an output pin
B	RW	INPUT			Connect or disconnect input buffer
			Connect	0	Connect input buffer
			Disconnect	1	Disconnect input buffer
C	RW	PULL			Pull configuration
			Disabled	0	No pull
			Pulldown	1	Pull down on pin
			Pullup	3	Pull up on pin
D	RW	DRIVE			Drive configuration
			S0S1	0	Standard '0', standard '1'
			H0S1	1	High drive '0', standard '1'
			S0H1	2	Standard '0', high drive '1'
			H0H1	3	High drive '0', high 'drive '1''
			D0S1	4	Disconnect '0' standard '1' (normally used for wired-or connections)
			D0H1	5	Disconnect '0', high drive '1' (normally used for wired-or connections)
			S0D1	6	Standard '0'. disconnect '1' (normally used for wired-and connections)
			H0D1	7	High drive '0', disconnect '1' (normally used for wired-and connections)
E	RW	SENSE			Pin sensing mechanism
			Disabled	0	Disabled
			High	2	Sense for high level
			Low	3	Sense for low level

Writing to arbitrary bits

- Remember that you can't just write to arbitrary bits
  - You'll have to use bit-operations to do so
  - In C: `&`, `|`, `~`, `^`, `<<`, `>>`

- For a review of "bit masking" operations, see bonus slides

Outline

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

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
    - Any pin in the Port can trigger the interrupt
    - Software checks which pin(s) caused the event to occur
    - Very low power operation (works with system clocks off)
- Unclear to me 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)

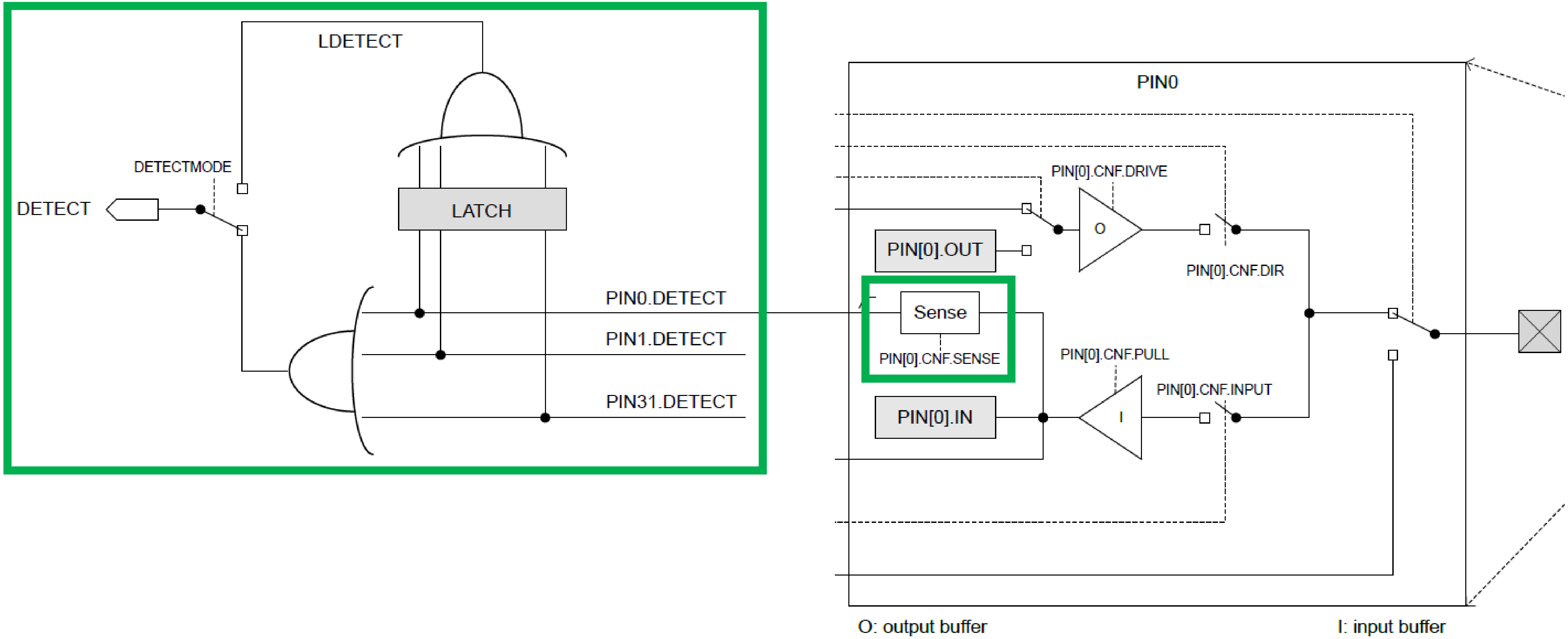
- Configure it
  - Port and Pin number
  - Task (output), Event (input), or Disabled
  - Polarity for input events
    - Low-to-high
    - High-to-low
    - Toggle (both directions)

- 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

Outline

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

- Bonus: Bit Masking

Bit Masking

- How do you manipulate certain bits within a number?

- Combines some of the ideas we've already learned
  - ~, &, |, <<, >>

- Steps
  1. Create a "bit mask" which is a pattern to choose certain bits
  2. Use & or | to combine it with your number
  3. Optional: Use >> to move the bits to the least significant position

Bit mask values

- Selecting bits, use the AND operation
  - 1 means to select that bit
  - 0 means to not select that bit

Select bottom four bits:

```
num & 0x0F
```

- Writing bits
  - Writing a one, use the OR operation
    - 1 means to write a one to that position
    - 0 is unchanged
  - Writing a zero, use the AND operation
    - 0 means to write a zero to that position
    - 1 is unchanged

Set 6th bit to one:

```
num | (1 << 6)
num | (0b01000000)
```

Clear 6th bit to zero:

```
num & (~(1 << 6))
num & (~(0b01000000))
num & (0b10111111)
```

Example: swap nibbles in byte

- Nibble - 4 bits (one hexit)
  - Input: 0x4F -> Output 0xF4
  - Method:
    - 1. Shift and select upper four bits
    - 2. Shift and select lower four bits
    - 3. Combine the two nibbles

What are the values of the new upper bits?

Unsigned -> Will be zero

```
uint8_t lower = input >> 4;
uint8_t upper = input << 4;
uint8_t output = upper | lower; // combines two halves
```

Shifting implicitly zero'd out irrelevant bits.
Otherwise we would have needed an & operation too.

Example: selecting bits

- Select bits 2 and 3 from a number

```
Input: 0b01100100
 Mask: 0b00001100
```

```
  0b01100100
& 0b00001100
------------
  0b00000100
```

Finally, shift right by two to get the values in the least significant position:

```
0b00000001
```

In C:

```
result = (input & 0x0C) >> 2;
```