

What is an embedded system?

- Embedded system is a computer based system that is designed for a single purpose or limited set of purposes
- The key difference between an embedded system and a general purpose computer is in the available programs. An embedded system runs software that is provided by the system manufacturer. In a general purpose computer the user can decide the which programs and from which vendor to run
- The application of the embedded system plays a key role in programming
 - Embedded system is (typically) run on its own without an operator
 - Reliability, fault tolerance and fault recovery are important
 - It is not always possible to shutdown or reboot the system at any time
 - Safety (e.g. patient, operator, service personnel, etc.)
 - Interacts with its environment
 - Sensors, actuators, communication interfaces, etc.
 - Can be a part of a larger system

Embedded systems programming

- Software is not (typically) developed on the target processor – target processor is the one that is going to run the application
 - Separate development environment adds more challenge to testing
 - It isn't always possible to simulate all subsystems but the tests must be run on the target hardware
 - Must consider repeatability and easy addition of test cases
- The resources of an embedded system are often sized to meet the application requirements
 - Aim for low unit cost → constrained resources (RAM, computing power, power consumption, ...)
 - Constraints must be taken into account in development

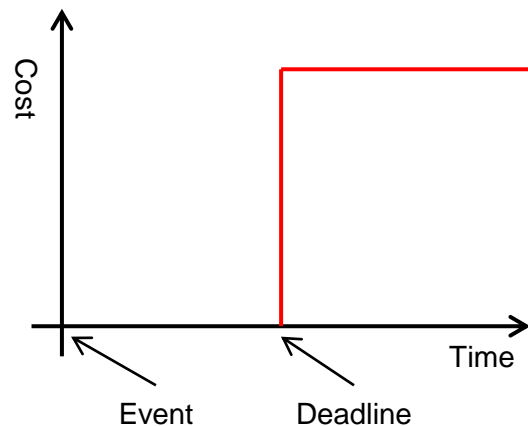
Embedded systems programming

- Embedded systems are often real time systems as well
 - Timing constraints can be hard or soft
 - Response time is a very typical constraint
 - Response to an event or message processing time
- Hard realtime
 - Failure to meet the requirements is always a critical error
- Soft realtime
 - Failure to meet the deadline is undesirable but not necessarily an error condition
 - Typically statistical figures
 - For example average processing time or processing time variance

Real time systems

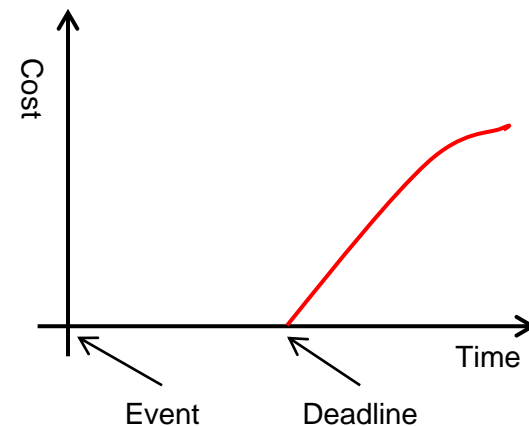
Hard real time

- Failure to meet the deadline causes danger or major financial damage



Soft real time

- Failure to meet the deadline increases cost but is acceptable
- Often related to Quality of Service (QoS)



Timing requirements

- Embedded systems react to events
 - Messages
 - Signals (external signals)
 - Timers
 - Integrated peripherals
 - Communication with other systems (e.g. UART, network card, etc.)
- Events have a hierarchy
 - Layers of protocol stack
- Arrival models (for event)
 - Periodical
 - Some variation may be present in the period
 - Non-periodical
 - For example burst transmission

Timing requirements

- Execution/processing times of events play a key role in evaluating the system response time
 - Hard realtime systems are evaluated using the worst-case execution times
 - Soft real time systems are evaluated using average execution times
 - Variance is typically an important QoS parameter
- Event and actions can occur sequentially or simultaneously
 - Sequential and independent actions and events are "easy"
 - Codependent simultaneous actions and events require synchronization

Special characteristics of embedded programming

- Pointers to control registers of peripherals and processor hardware
 - Actions are performed or events handled by reading and/or writing processor and peripherals control registers
 - Ordering and timing of read and write is critical
 - Control registers are usually mapped in to the address space and they are read/written the same way as RAM
 - Programmer needs an understanding of the hardware
 - The addresses of control registers can be found in manufacturers documentation
- Volatile modifier
 - Tells the compiler that the value of a variable can change at any time and in a way that compiler can not detect – even when the variable is not being accessed in the program
 - Pointers to memory mapped control registers are declared with volatile modifier

Special characteristics of embedded programming

- Bit manipulation
 - Logical operations and shifts are needed in embedded programming to allow access to individual bits or groups of bits in the control registers
- Lower hardware abstraction level
 - Programmer needs to know how the hardware works
 - Operating systems are becoming more and more common in mid- and high end embedded systems and hide some of the details behind OS API
 - Device driver writer/system programmer still needs to understand the hardware
 - Embedded/real time operating systems tend to be "lighter" than general purpose computer OSes
- Interrupts
- Low level code is often written in C
 - C++ programs can reuse C code or encapsulate it in an object
 - Possibility of using C++ must be taken into account in C code

Special characteristics of embedded programming

- Resource constrains
 - Low end embedded systems have quite limited amount of RAM
 - Frequent use of malloc/free can cause problems (or use of new/delete when programming in C++)
 - Hardware/peripherals are physical resources that can't be duplicated → locking is needed in multithreaded/interrupt driven software
- 24/7 unsupervised operation requires special attention in memory management
 - Even a small memory leak will crash the system after a while

Bit manipulation

- IO devices are commonly mapped into memory locations
 - Single memory location contains up to 32 bits that control an IO device
- A bit or group of bit within for example 32 bit word can be manipulated with shift and mask operations
 - Shifting allows us to keep the values in smaller and usually more intuitive range
- For example change bits 13-15 within a 32-bit word without changing the other bits. SFR is the 32-bit word to modify, val is an integer value in the range 0 – 7
 - $SFR = (SFR \& 0xFFFF1FFF) \mid (val \ll 13);$or
 - $SFR = (SFR \& \sim 0xE000) \mid (val \ll 13);$or
 - $SFR \&= \sim 0xE000;$
 $SFR \mid= val \ll 13;$

```
void gpio_set_pulls(uint gpio, bool up, bool down) {
    check_gpio_param(gpio);
    hw_write_masked(
        addr: &padsbank0_hw->io[gpio],
        values: (bool_to_bit(up) << PADS_BANK0_GPI00_PUE_LSB) | (bool_to_bit(down) << PADS_BANK0_GPI00_PDE_LSB),
        write_mask: PADS_BANK0_GPI00_PUE_BITS | PADS_BANK0_GPI00_PDE_BITS
    );
}
```

Memory mapped devices and typecasting

```
#define IO_QSPI_BASE _u(0x40018000)
#define PADS_BANK0_BASE _u(0x4001c000)
#define PADS_QSPI_BASE _u(0x40020000)
```

```
typedef struct {
    _REG_(PADS_BANK0_VOLTAGE_SELECT_OFFSET) // PADS_BANK0_VOLTAGE_SELECT
    // Voltage select
    // 0x00000001 [0] : VOLTAGE_SELECT (0)
    io_rw_32 voltage_select;

    _REG_(PADS_BANK0_GPIO0_OFFSET) // PADS_BANK0_GPIO0
    // (Description copied from array index 0 register PADS_BANK0_GPIO0 applies similarly to other array indexes)
    //
    // Pad control register
    // 0x00000080 [7] : OD (0): Output disable
    // 0x00000040 [6] : IE (1): Input enable
    // 0x00000030 [5:4] : DRIVE (1): Drive strength
    // 0x00000008 [3] : PUE (0): Pull up enable
    // 0x00000004 [2] : PDE (1): Pull down enable
    // 0x00000002 [1] : SCHMITT (1): Enable schmitt trigger
    // 0x00000001 [0] : SLEWFAST (0): Slew rate control
    io_rw_32 io[NUM_BANK0_GPIOS]; // 30
} padsbank0_hw_t;
```

The mapping between the members of a structure and the memory locations is compiler dependent.

Table 282. List of IO_BANK0 registers

The User Bank IO registers start at a base address of 0x40014000 (defined as IO_BANK0_BASE in SDK).

Offset	Name	Info
0x000	GPIO0_STATUS	GPIO status
0x004	GPIO0_CTRL	GPIO control including function select and overrides.
0x008	GPIO1_STATUS	GPIO status
0x00c	GPIO1_CTRL	GPIO control including function select and overrides.
0x010	GPIO2_STATUS	GPIO status
0x014	GPIO2_CTRL	GPIO control including function select and overrides.
0x018	GPIO3_STATUS	GPIO status
0x01c	GPIO3_CTRL	GPIO control including function select and overrides.

From the user manual

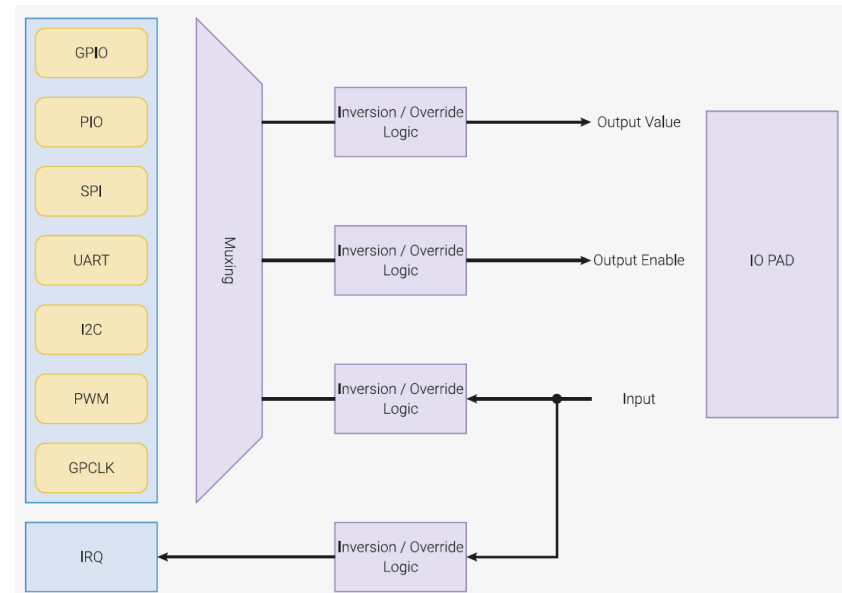
IO-interfaces

- Simple IO-interfaces are passive and always ready for data transfer (for example buttons, switches, leds, etc.)
 - Read or write can be performed at any time
- Sophisticated IO-interfaces are not always ready for data transfer since they may be busy with ongoing activity (for example still transmitting previous data)
 - Processor needs to be notified when interface is ready for transfer (for example when UART is ready to transmit new byte)
 - Interface needs to be configured before data transfer (for example set UART baud rate or frame size of network adapter)
 - State of interface can read from status register(s).
 - Status registers are mapped into RAM or IO-address space

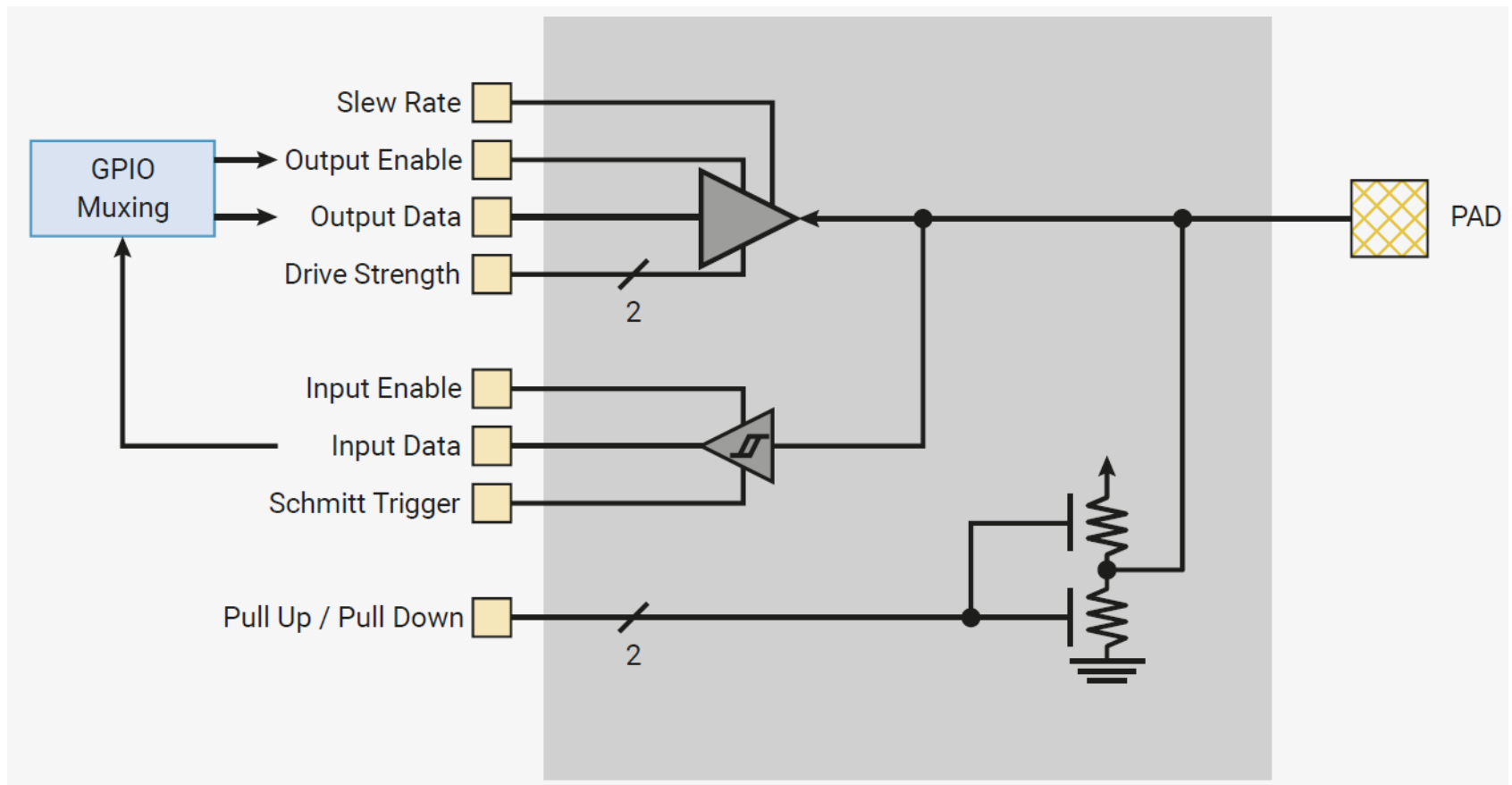
RP2040 GPIO pins are configurable

- Digital pins
 - Output pins
 - Input pins
 - Pullup
 - Pulldown
 - High impedance (no pullup/pulldown)
 - Repeater
- Analog pins (not available on all pins)
- Special function
 - Pin is controlled by a peripheral
 - Some peripherals are available only on limited pins

	Function								
GPIO	F1	F2	F3	F4	F5	F6	F7	F8	F9
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PI00	PI01		USB OVCCR DET
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PI00	PI01		USB VBUS DET
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PI00	PI01		USB VBUS EN
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PI00	PI01		USB OVCCR DET
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PI00	PI01		USB VBUS DET
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PI00	PI01		USB VBUS EN
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PI00	PI01		USB OVCCR DET
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PI00	PI01		USB VBUS DET
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PI00	PI01		USB VBUS EN
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PI00	PI01		USB OVCCR DET

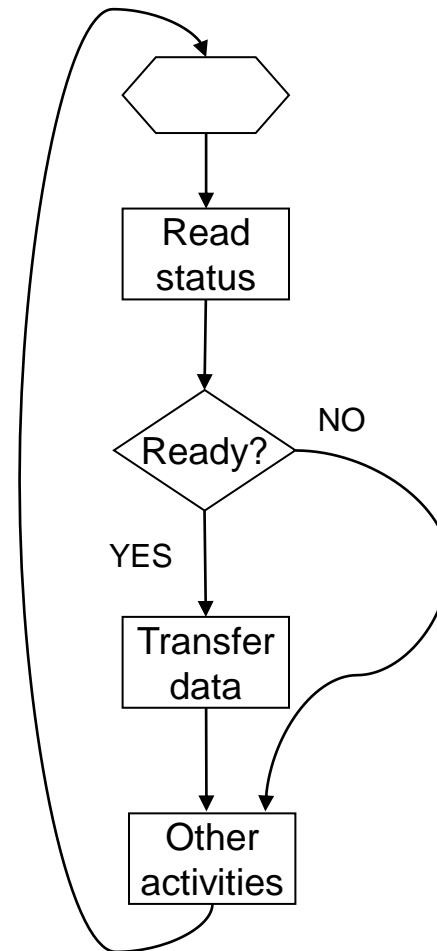


GPIO Pads



Programmed IO

- Polling
 - Read interface status
 - If interface is ready then transfer data
 - If interface is not ready continue with other activities
 - Read interface status...
- Other activities can delay interface access (data can be lost or transfer speed is decreased)
- Polling wastes processor resources if device is not ready

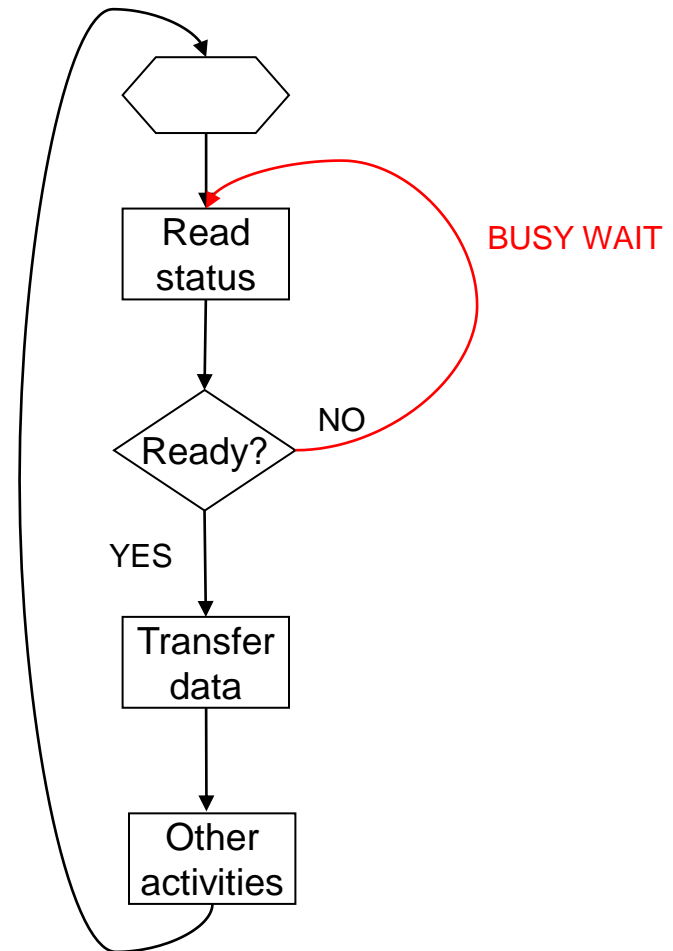


Button polling example

```
while(1) {  
    if(gpio_get(22)) {  
        SetValue(7);  
    }  
    if(gpio_get(21)) {  
        SetValue(0);  
    }  
    if(!gpio_get(20) {  
        SetValue(counter);  
    }  
}
```



Programmed IO

- Busy wait stops other activities and continues only after interface is ready
- Slows down other activities
- Practical if
 - There is only one IO-device
- or
- Execution may not continue until the device is ready (for example wait until initial configuration is done)
- or
- Your applications response time is still acceptable



Busy wait loop

```
while(1) {  
    if(gpio_get(22)) {  
        SetValue(0);  
        while(gpio_get(22));  
        SetValue(counter);  
    }  
}
```



- In this example the loop body is an empty statement
 - The loop runs the test over and over again at high speed until the button is released

A little sidetrack to switch bounce

- Buttons typically exhibit switch bounce when the switch is closed
- When a button is pressed it takes a short while for contacts to settle and during that time the contacts may open and close multiple times
- Without filtering the switch bounce may be counted as multiple key presses
- The simplest way of filtering is to wait for typical switch bounce time after a detected key press until the switch state is checked again
 - The switch bounce varies – usually switches from the same batch exhibit similar behavior
 - Can be anything from none to 150 ms – usually in the order of couple of milliseconds (but don't count on that)
- If response time is not a critical issue a busy loop can be used in filtering



Switch bounce on an input with a pull up resistor

Busy loop switch bounce filter

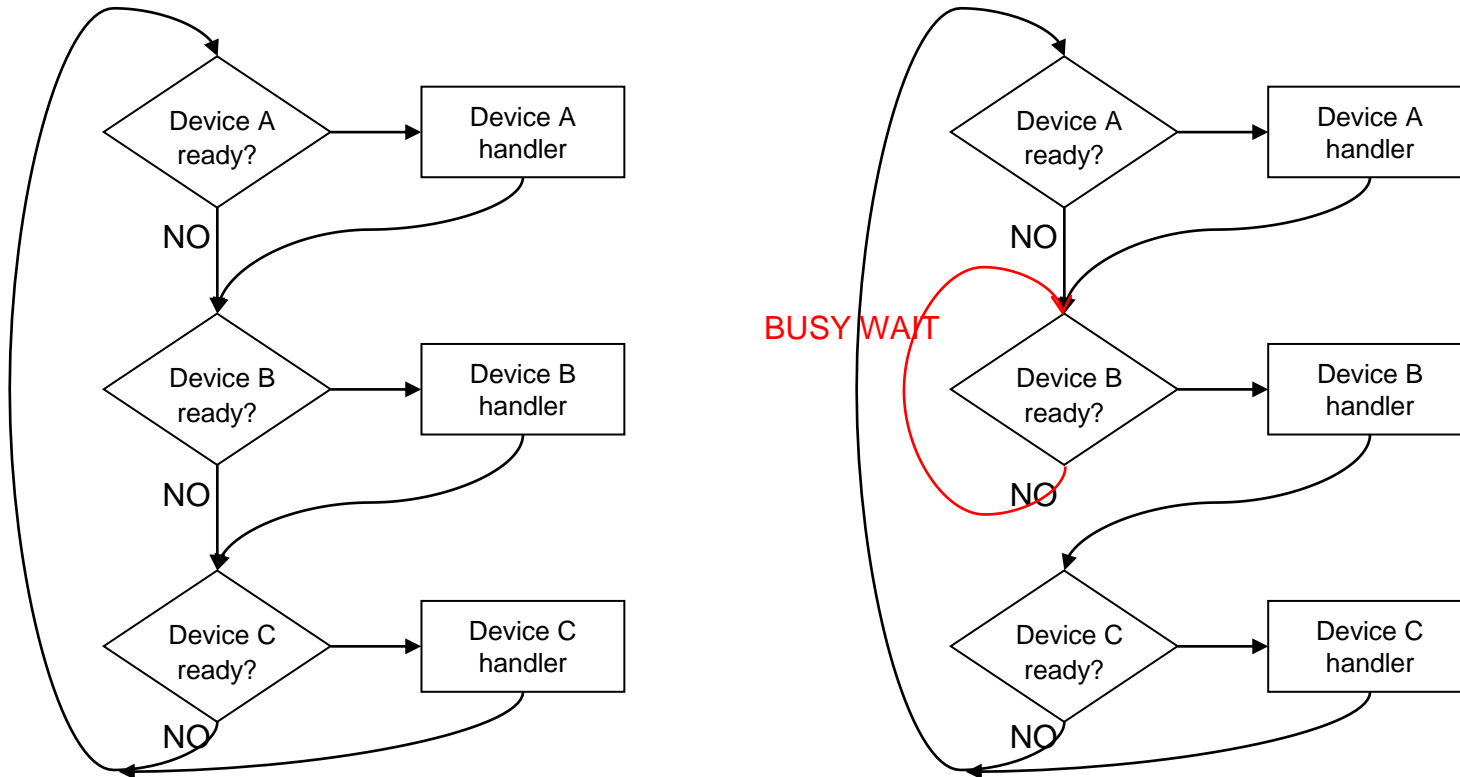
```
bool pressed(void)
{
    int press = 0;
    int release = 0;
    while(press < 3 && release < 3){
        if(gpio_get(22)) {
            press++;
            release = 0;
        }
        else {
            release++;
            press = 0;
        }
        sleep_ms(10); // wait 10 ms
    }
    if(press > release) return true;
    else return false;
}
```

This loop runs until we have read the same value three times in a row

Wait a while to allow switch to stabilize

What is the minimum execution time of this function?
What is the maximum execution time?

Programmed IO



If you have more than one device a busy wait stops the polling loop! Even a modest delay in a polling loop can have a huge impact on the responsiveness of the program.

Programmed IO

- Example
 - USB keyboard is attached to a microcontroller that runs at 72 MHz
 - USB HID transfer rate is max 64 kbps (8 kilobytes per second)
 - If polling takes 220 clock cycles and one poll can read one character we spend about 2.5 percent of the CPU time for polling
 - If user types twelve eighth character words per minute (~100 chars per minute) we still need to poll 8192×60 times per minute to ensure that all characters are read.
 - Yet 99,98% of pollings return no data!

$$\frac{8192 \times 220}{72 \times 10^6} \approx 2,5 \%$$

Note: In a real world a device with this high transfer rate would never be implemented with polling