



Memory Mapped IO used for GPIO

Lecture 2



GPIO for RP2040

- Memory Mapped I/O
 - GPIO Peripheral
- Embedded Rust Stack
- embassy-rs



MMIO

Memory Mapped Input Output



8 bit processor

a simple 8 bit processor with a text display





STM32L0x2

A real MCU

| | |
|---------------------------|--|
| Cortex-M0+ Peripherals | MCU's <i>settings</i> and internal peripherals, available at the same address on all M0+ |
| Peripherals | GPIO, USART, SPI, I2C, USB, etc |
| Flash | The storage space |
| SRAM | RAM memory |
| @0x0000_0000 | Alias for SRAM or Flash |





System Control Registers

@0xe000_0000

Compute the actual address

- $0xe000_0000 + \text{Offset}$

Examples:

- SYST_CSR: **0xe000_e010** ($0xe000_0000 + 0xe010$)
- CPUID: **0xe000_ed00** ($0xe000_0000 + 0xed00$)

```
1  const SYS_CTRL: usize = 0xe000_0000;  
2  const CPUID: usize = 0xed00;  
3  
4  let cpuid_reg = (SYS_CTRL + CPUID) as *const u32;  
5  let cpuid_value = unsafe { *cpuid_reg };
```

⚠ Processors use cache!

| Offset | Name | Info |
|--------|----------------------------|--|
| 0xe010 | SYST_CSR | SysTick Control and Status Register |
| 0xe014 | SYST_RVR | SysTick Reload Value Register |
| 0xe018 | SYST_CVR | SysTick Current Value Register |
| 0xe01c | SYST_CALIB | SysTick Calibration Value Register |
| 0xe100 | NVIC_ISER | Interrupt Set-Enable Register |
| 0xe180 | NVIC_ICER | Interrupt Clear-Enable Register |
| 0xe200 | NVIC_ISPR | Interrupt Set-Pending Register |
| 0xe280 | NVIC_ICPR | Interrupt Clear-Pending Register |
| 0xe400 | NVIC_IPR0 | Interrupt Priority Register 0 |
| 0xe404 | NVIC_IPR1 | Interrupt Priority Register 1 |
| 0xe408 | NVIC_IPR2 | Interrupt Priority Register 2 |
| 0xe40c | NVIC_IPR3 | Interrupt Priority Register 3 |
| 0xe410 | NVIC_IPR4 | Interrupt Priority Register 4 |
| 0xe414 | NVIC_IPR5 | Interrupt Priority Register 5 |
| 0xe418 | NVIC_IPR6 | Interrupt Priority Register 6 |
| 0xe41c | NVIC_IPR7 | Interrupt Priority Register 7 |
| 0xed00 | CPUID | CPUID Base Register |
| 0xed04 | ICSR | Interrupt Control and State Register |
| 0xed08 | VTOR | Vector Table Offset Register |
| 0xed0c | AIRCR | Application Interrupt and Reset Control Register |
| 0xed10 | SCR | System Control Register |
| 0xed14 | CCR | Configuration and Control Register |



8 bit processor

with cache





Cache Write-Trough

CPUID: `0xe000_ed00` (`0xe000_0000 + 0xed00`)

```
1 use core::ptr::read_volatile;
2
3 const SYS_CTRL: usize = 0xe000_0000;
4 const CPUID: usize = 0xed00;
5
6 let cpuid_reg = (SYS_CTRL + CPUID) as *const u32;
7 unsafe {
8     read_volatile(cpuid_reg) // avoid cache
9 }
```

`read_volatile`,
`write_volatile`

no cache or compiler
optimization

`read`, `write`, `*p`

use cache and compiler
optimization

| Offset | Name | Info |
|--------|----------------------------|--|
| 0xe010 | SYST_CSR | SysTick Control and Status Register |
| 0xe014 | SYST_RVR | SysTick Reload Value Register |
| 0xe018 | SYST_CVR | SysTick Current Value Register |
| 0xe01c | SYST_CALIB | SysTick Calibration Value Register |
| 0xe100 | NVIC_IJSER | Interrupt Set-Enable Register |
| 0xe180 | NVIC_ICER | Interrupt Clear-Enable Register |
| 0xe200 | NVIC_ISPR | Interrupt Set-Pending Register |
| 0xe280 | NVIC_ICPR | Interrupt Clear-Pending Register |
| 0xe400 | NVIC_IPR0 | Interrupt Priority Register 0 |
| 0xe404 | NVIC_IPR1 | Interrupt Priority Register 1 |
| 0xe408 | NVIC_IPR2 | Interrupt Priority Register 2 |
| 0xe40c | NVIC_IPR3 | Interrupt Priority Register 3 |
| 0xe410 | NVIC_IPR4 | Interrupt Priority Register 4 |
| 0xe414 | NVIC_IPR5 | Interrupt Priority Register 5 |
| 0xe418 | NVIC_IPR6 | Interrupt Priority Register 6 |
| 0xe41c | NVIC_IPR7 | Interrupt Priority Register 7 |
| 0xed00 | CPUID | CPUID Base Register |
| 0xed04 | ICSR | Interrupt Control and State Register |
| 0xed08 | VTOR | Vector Table Offset Register |
| 0xed0c | AIRCR | Application Interrupt and Reset Control Register |
| 0xed10 | SCR | System Control Register |
| 0xed14 | CCR | Configuration and Control Register |



Read the CPUID

About the MCU

```
1 use core::ptr::read_volatile;
2
3 const SYS_CTRL: usize = 0xe000_0000;
4 const CPUID: usize = 0xed00;
5
6 let cpuid_reg = (SYS_CTRL + CPUID) as *const u32;
7 let cpuid_value = unsafe {
8     read_volatile(cpuid_reg)
9 };
10
11 // shift right 24 bits and keep only the last 8 bits
12 let variant = (cpuid_value >> 24) & 0b1111_1111;
13
14 // shift right 16 bits and keep only the last 4 bits
15 let architecture = (cpuid_value >> 16) & 0b1111;
16
17 // shift right 4 bits and keep only the last 12 bits
18 let part_no = (cpuid_value >> 4) & 0b11_1111_1111;
19
20 // shift right 0 bits and keep only the last 4 bits
21 let revision = (cpuid_value >> 0) & 0b1111;
```

CPUID Register

Offset: 0xed04

| Bits | Name | Description | Type | Reset |
|-------|--------------|--|------|-------|
| 31:24 | IMPLEMENTER | Implementor code: 0x41 = ARM | RO | 0x41 |
| 23:20 | VARIANT | Major revision number n in the rnpn revision status: 0x0 = Revision 0. | RO | 0x0 |
| 19:16 | ARCHITECTURE | Constant that defines the architecture of the processor: 0xC = ARMv6-M architecture. | RO | 0xc |
| 15:4 | PARTNO | Number of processor within family: 0xC60 = Cortex-M0+ | RO | 0xc60 |
| 3:0 | REVISION | Minor revision number m in the rnpn revision status: 0x1 = Patch 1. | RO | 0x1 |



AIRCR

Application Interrupt and Reset Control Register

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const SYS_CTRL: usize = 0xe000_0000;
5 const AIRCR: usize = 0xed0c;
6
7 const VECTKEY: u32 = 16;
8 const SYSRESETREQ: u32 = 2;
9
10 let aircr_register = (SYS_CTRL + AIRCR) as *mut u32;
11 let mut aircr_value = unsafe {
12     read_volatile(aircr_register)
13 };
14
15 aircr_value = aircr_value & ~(0x1111 << VECTKEY);
16 aircr_value = aircr_value | (0x05fa << VECTKEY);
17 aircr_value = aircr_value | (1 << SYSRESETREQ);
18
19 unsafe {
20     write_volatile(aircr_register, aircr_value);
21 }
```

AIRCR Register

Offset: 0xed0c

| Bits | Name | Description | Type | Reset |
|-------|-----------|--|------|--------|
| 31:16 | VECTKEY | Register key: Reads as Unknown On writes, write 0x05FA to VECTKEY, otherwise the write is ignored. | RW | 0x0000 |
| 15 | ENDIANESS | Data endianness implemented: 0 = Little-endian. | RO | 0x0 |
| 14:3 | Reserved. | - | - | - |

| Bits | Name | Description | Type | Reset |
|------|---------------|---|------|-------|
| 2 | SYSRESETREQ | Writing 1 to this bit causes the SYSRESETREQ signal to the outer system to be asserted to request a reset. The intention is to force a large system reset of all major components except for debug. The C_HALT bit in the DHCSR is cleared as a result of the system reset requested. The debugger does not lose contact with the device. | RW | 0x0 |
| 1 | VECTCLRACTIVE | Clears all active state information for fixed and configurable exceptions. This bit: is self-clearing, can only be set by the DAP when the core is halted. When set: clears all active exception status of the processor, forces a return to Thread mode, forces an IPSR of 0. A debugger must re-initialize the stack. | RW | 0x0 |
| 0 | Reserved. | - | - | - |



Read and Write

they do stuff

- Read
 - reads the value of a register
 - might ask the peripheral to do something
- Write
 - writes the value to a register
 - might ask the peripheral to do something
 - SYSRESETREQ

AIRCR Register

Offset: 0xed0c

| Bits | Name | Description | Type | Reset |
|-------|-----------|--|------|--------|
| 31:16 | VECTKEY | Register key: Reads as Unknown On writes, write 0x05FA to VECTKEY, otherwise the write is ignored. | RW | 0x0000 |
| 15 | ENDIANESS | Data endianness implemented: 0 = Little-endian. | RO | 0x0 |
| 14:3 | Reserved. | - | - | - |

| Bits | Name | Description | Type | Reset |
|------|---------------|---|------|-------|
| 2 | SYSRESETREQ | Writing 1 to this bit causes the SYSRESETREQ signal to the outer system to be asserted to request a reset. The intention is to force a large system reset of all major components except for debug. The C_HALT bit in the DHCSR is cleared as a result of the system reset requested. The debugger does not lose contact with the device. | RW | 0x0 |
| 1 | VECTCLRACTIVE | Clears all active state information for fixed and configurable exceptions. This bit: is self-clearing, can only be set by the DAP when the core is halted. When set: clears all active exception status of the processor, forces a return to Thread mode, forces an IPSR of 0. A debugger must re-initialize the stack. | RW | 0x0 |
| 0 | Reserved. | - | - | - |



SVD XML File

System View Description

```
1  <device schemaVersion="1.1"
2    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" xs:noNamespaceSchemaLocation="CMSIS-SVD.xsd">
3    <name>RP2040</name>
4    <peripherals>
5      <name>PPB</name>
6      <baseAddress>0xe0000000</baseAddress>
7      <register>
8        <name>CPUID</name>
9        <addressOffset>0xed00</addressOffset>
10       <resetValue>0x410cc601</resetValue>
11       <fields>
12         <field>
13           <name>IMPLEMENTER</name>
14           <description>Implementor code: 0x41 = ARM</description>
15           <bitRange>[31:24]</bitRange>
16           <access>read-only</access>
17         </field>
18         <!-- rest of the fields of the register -->
19       </fields>
20     </register>
21   </peripherals>
22 </device>
```



Bitwise Ops

How to set and clear bits



Set bit

set the 1 on position bit of register

```
1 fn set_bit(register: usize, bit: u8) -> usize {
2     // assume register is 0b1000, bit is 2
3     // 1 << 2 is 0b0100
4     // 0b1000 | 0b0100 is 0b1100
5     register | 1 << bit
6 }
```

Set multiple bits

```
1 fn set_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1000, bits is 0b0111
3     // 0b1000 | 0b0111 is 0b1111
4     register | bits
5 }
```



Clear bit

set the 0 on position bit of register

```
1 fn clear_bit(register: usize, bit: u8) -> usize {
2     // assume register is 0b1100, bit is 2
3     // 1 << 2 is 0b0100
4     // !(1 << 3) is 0b1011
5     // 0b1100 & 0b1011 is 0b1000
6     register & !(1 << bit)
7 }
```

Clear multiple bits

```
1 fn clear_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1111, bits is 0b0111
3     // !bits = 0b1000
4     // 0b1111 & 0b1000 is 0b1000
5     register & !bits
6 }
```



Flip bit

flip the bit on position `bit` of `register`

```
1  fn flip_bit(register: usize, bit: u8) -> usize {
2      // assume register is 0b1000, bit is 2
3      // 1 << 2 is 0b0100
4      // 0b1100 ^ 0b0100 is 0b1000
5      register ^ 1 << bit
6  }
```

Flip multiple bits

```
1  fn flip_bits(register: usize, bits: usize) -> usize {
2      // assume register is 0b1000, bits is 0b0111
3      // 0b1000 ^ 0b0111 is 0b1111
4      register ^ bits
5  }
```




GPIO

General Purpose Input Output for RP2040



Bibliography

for this section

Raspberry Pi Ltd, *RP2040 Datasheet*

- Chapter 2 - *System Description*
 - Section 2.3 - *Processor subsystem*
 - Subsection 2.3.1 - *SIO*
 - Subsection 2.3.1.2 - *GPIO Control*
 - Section 2.4 - *Cortex-M0+* (except NVIC and MPU)
 - Section 2.19 - *GPIO* (except Interrupts)



GPIO

Peripherals



SIO Single Cycle Input/Output, is able to control the GPIO pins

GPIO Multiplexes the functions of the GPIO pins

| | Function | | | | | | | | |
|------|----------|-----------|----------|--------|-----|------|------|----|---------------|
| GPIO | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 |
| 0 | SPI0 RX | UART0 TX | I2C0 SDA | PWM0 A | SIO | PIO0 | PIO1 | | USB OVCCR DET |
| 1 | SPI0 CSn | UART0 RX | I2C0 SCL | PWM0 B | SIO | PIO0 | PIO1 | | USB VBUS DET |
| 2 | SPI0 SCK | UART0 CTS | I2C1 SDA | PWM1 A | SIO | PIO0 | PIO1 | | USB VBUS EN |
| 3 | SPI0 TX | UART0 RTS | I2C1 SCL | PWM1 B | SIO | PIO0 | PIO1 | | USB OVCCR DET |
| 4 | SPI0 RX | UART1 TX | I2C0 SDA | PWM2 A | SIO | PIO0 | PIO1 | | USB VBUS DET |
| 5 | SPI0 CSn | UART1 RX | I2C0 SCL | PWM2 B | SIO | PIO0 | PIO1 | | USB VBUS EN |
| 6 | SPI0 SCK | UART1 CTS | I2C1 SDA | PWM3 A | SIO | PIO0 | PIO1 | | USB OVCCR DET |
| 7 | SPI0 TX | UART1 RTS | I2C1 SCL | PWM3 B | SIO | PIO0 | PIO1 | | USB VBUS DET |
| 8 | SPI1 RX | UART1 TX | I2C0 SDA | PWM4 A | SIO | PIO0 | PIO1 | | USB VBUS EN |
| 9 | SPI1 CSn | UART1 RX | I2C0 SCL | PWM4 B | SIO | PIO0 | PIO1 | | USB OVCCR DET |
| 10 | SPI1 SCK | UART1 CTS | I2C1 SDA | PWM5 A | SIO | PIO0 | PIO1 | | USB VBUS DET |
| 11 | SPI1 TX | UART1 RTS | I2C1 SCL | PWM5 B | SIO | PIO0 | PIO1 | | USB VBUS EN |
| 12 | SPI1 RX | UART0 TX | I2C0 SDA | PWM6 A | SIO | PIO0 | PIO1 | | USB OVCCR DET |
| 13 | SPI1 CSn | UART0 RX | I2C0 SCL | PWM6 B | SIO | PIO0 | PIO1 | | USB VBUS DET |
| 14 | SPI1 SCK | UART0 CTS | I2C1 SDA | PWM7 A | SIO | PIO0 | PIO1 | | USB VBUS EN |

IO Bank (GPIO): Use the correct MUX function (F5)

SIO: Set the pin as Input or Output



SIO Registers

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

| Offset | Name | Info |
|--------|------------------------------|---------------------------|
| 0x000 | CPUID | Processor core identifier |
| 0x004 | GPIO_IN | Input value for GPIO pins |
| 0x008 | GPIO_HI_IN | Input value for QSPI pins |
| 0x010 | GPIO_OUT | GPIO output value |
| 0x014 | GPIO_OUT_SET | GPIO output value set |
| 0x018 | GPIO_OUT_CLR | GPIO output value clear |
| 0x01c | GPIO_OUT_XOR | GPIO output value XOR |
| 0x020 | GPIO_OE | GPIO output enable |
| 0x024 | GPIO_OE_SET | GPIO output enable set |
| 0x028 | GPIO_OE_CLR | GPIO output enable clear |

- Input
 - set `GPIO_OE` bit x to 0
 - read `GPIO_IN` bit x
- Output
 - set `GPIO_OE` bit x to 1
 - write `GPIO_OUT` bit x

GPIO_OE

| Bits | Description | Type | Reset |
|-------|--|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to <code>GPIO_OE</code> simultaneously (or to a <code>SET/CLR/XOR</code> alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result. | RW | 0x00000000 |

GPIO_IN

| Bits | Description | Type | Reset |
|-------|----------------------------|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Input value for GPIO0...29 | RO | 0x00000000 |

GPIO_OUT

| Bits | Description | Type | Reset |
|-------|---|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Set output level (1/0 → high/low) for GPIO0...29. Reading back gives the last value written, NOT the input value from the pins. If core 0 and core 1 both write to <code>GPIO_OUT</code> simultaneously (or to a <code>SET/CLR/XOR</code> alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result. | RW | 0x00000000 |



SIO Input

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

| Offset | Name | Info |
|--------|---------------------------|---------------------------|
| 0x000 | <code>CPUID</code> | Processor core identifier |
| 0x004 | <code>GPIO_IN</code> | Input value for GPIO pins |
| 0x008 | <code>GPIO_HI_IN</code> | Input value for QSPI pins |
| 0x010 | <code>GPIO_OUT</code> | GPIO output value |
| 0x014 | <code>GPIO_OUT_SET</code> | GPIO output value set |
| 0x018 | <code>GPIO_OUT_CLR</code> | GPIO output value clear |
| 0x01c | <code>GPIO_OUT_XOR</code> | GPIO output value XOR |
| 0x020 | <code>GPIO_OE</code> | GPIO output enable |
| 0x024 | <code>GPIO_OE_SET</code> | GPIO output enable set |
| 0x028 | <code>GPIO_OE_CLR</code> | GPIO output enable clear |

GPIO_OE

| Bits | Description | Type | Reset |
|-------|--|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result. | RW | 0x00000000 |

GPIO_IN

| Bits | Description | Type | Reset |
|-------|----------------------------|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Input value for GPIO0...29 | RO | 0x00000000 |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIO_OE: *mut u32 = 0xd000_0020 as *mut u32;
5 const GPIO_IN: *const u32 = 0xd000_0004 as *const u32;
6
7 let value = unsafe {
8     // write_volatile(GPIO_OE, !(1 << pin));
9     let gpio_oe = read_volatile(GPIO_OE);
10    // set bin `pin` of `gpio_oe` to 0 (input)
11    gpio_oe = gpio_oe & !(1 << pin);
12    write_volatile(GPIO_OE, gpio_oe);
13    read_volatile(GPIO_IN) >> pin & 0b1
14 };
```



SIO Input

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

| Offset | Name | Info |
|--------|---------------------------|---------------------------|
| 0x000 | <code>CPUID</code> | Processor core identifier |
| 0x004 | <code>GPIO_IN</code> | Input value for GPIO pins |
| 0x008 | <code>GPIO_HI_IN</code> | Input value for QSPI pins |
| 0x010 | <code>GPIO_OUT</code> | GPIO output value |
| 0x014 | <code>GPIO_OUT_SET</code> | GPIO output value set |
| 0x018 | <code>GPIO_OUT_CLR</code> | GPIO output value clear |
| 0x01c | <code>GPIO_OUT_XOR</code> | GPIO output value XOR |
| 0x020 | <code>GPIO_OE</code> | GPIO output enable |
| 0x024 | <code>GPIO_OE_SET</code> | GPIO output enable set |
| 0x028 | <code>GPIO_OE_CLR</code> | GPIO output enable clear |

GPIO_OE_SET

| Bits | Description | Type | Reset |
|-------|---|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Perform an atomic bit-clear on GPIO_OE, i.e. <code>GPIO_OE &= ~wdata</code> | WO | 0x00000000 |

GPIO_IN

| Bits | Description | Type | Reset |
|-------|----------------------------|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Input value for GPIO0...29 | RO | 0x00000000 |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIO_OE_CLR: *mut u32 = 0xd000_0028 as *mut u32;
5 const GPIO_IN: *const u32 = 0xd000_0004 as *const u32;
6
7 let value = unsafe {
8     // set bit `pin` of `GPIO_OE` to 0 (input)
9     write_volatile(GPIO_OE_CLR, 1 << pin);
10    read_volatile(GPIO_IN) >> pin & 0b1
11 };
```



SIO Output

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

| Offset | Name | Info |
|--------|------------------------------|---------------------------|
| 0x000 | CPUID | Processor core identifier |
| 0x004 | GPIO_IN | Input value for GPIO pins |
| 0x008 | GPIO_HI_IN | Input value for QSPI pins |
| 0x010 | GPIO_OUT | GPIO output value |
| 0x014 | GPIO_OUT_SET | GPIO output value set |
| 0x018 | GPIO_OUT_CLR | GPIO output value clear |
| 0x01c | GPIO_OUT_XOR | GPIO output value XOR |
| 0x020 | GPIO_OE | GPIO output enable |
| 0x024 | GPIO_OE_SET | GPIO output enable set |
| 0x028 | GPIO_OE_CLR | GPIO output enable clear |

GPIO_OE_CLR

| Bits | Description | Type | Reset |
|-------|---|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Perform an atomic bit-clear on GPIO_OE, i.e. <code>GPIO_OE &= ~wdata</code> | WO | 0x00000000 |

GPIO_OUT

| Bits | Description | Type | Reset |
|-------|---|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Set output level (1/0 → high/low) for GPIO0...29. Reading back gives the last value written, NOT the input value from the pins. If core 0 and core 1 both write to GPIO_OUT simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result. | RW | 0x00000000 |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIO_OE_SET: *mut u32 = 0xd000_0024 as *mut u32;
5 const GPIO_OUT: *mut u32 = 0xd000_0010 as *mut u32;
6
7 unsafe {
8     // set bit `pin` of GPIO_OE to 1 (output)
9     write_volatile(GPIO_OE_SET, 1 << pin);
10    // write_volatile(GPIO_OUT, (value & 0b1) << pin);
11    let gpio_out = read_volatile(GPIO_OUT);
12    gpio_out = gpio_out | (value & 0b1) << pin;
13    write_volatile(GPIO_OUT, gpio_out);
14 }
```



SIO Output

efficient

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

| Offset | Name | Info |
|--------|------------------------------|---------------------------|
| 0x000 | CPUID | Processor core identifier |
| 0x004 | GPIO_IN | Input value for GPIO pins |
| 0x008 | GPIO_HI_IN | Input value for QSPI pins |
| 0x010 | GPIO_OUT | GPIO output value |
| 0x014 | GPIO_OUT_SET | GPIO output value set |
| 0x018 | GPIO_OUT_CLR | GPIO output value clear |
| 0x01c | GPIO_OUT_XOR | GPIO output value XOR |
| 0x020 | GPIO_OE | GPIO output enable |
| 0x024 | GPIO_OE_SET | GPIO output enable set |
| 0x028 | GPIO_OE_CLR | GPIO output enable clear |

GPIO_OUT_SET

| Bits | Description | Type | Reset |
|-------|--|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Perform an atomic bit-set on GPIO_OUT, i.e. <code>GPIO_OUT = wdata</code> | WO | 0x00000000 |

GPIO_OUT_CLR

| Bits | Description | Type | Reset |
|-------|---|------|------------|
| 31:30 | Reserved. | - | - |
| 29:0 | Perform an atomic bit-clear on GPIO_OUT, i.e. <code>GPIO_OUT &= ~wdata</code> | WO | 0x00000000 |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIO_OE_SET: *mut u32 = 0xd000_0024 as *mut u32;
5 const GPIO_OUT_SET: *mut u32 = 0xd000_0014 as *mut u32;
6 const GPIO_OUT_CLR: *mut u32 = 0xd000_0018 as *mut u32;
7
8 unsafe {
9     write_volatile(GPIO_OE_SET, 1 << pin);
10    let reg = match value {
11        0 => GPIO_OUT_CLR,
12        _ => GPIO_OUT_SET
13    };
14    write_volatile(reg, 1 << pin);
15 }
```




IO Bank



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. |

- set **FUNCSEL** to **5** (*SIO*)

GPIOx_CTRL

Offset: 0x004, 0x00c, ... 0x0ec ($0x4 + 8 * x$)

| Bits | Name | Description | Type | Reset |
|-------|-----------|--|------|-------|
| 31:30 | Reserved. | - | - | - |
| 29:28 | IRQOVER | 0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high | RW | 0x0 |
| 27:18 | Reserved. | - | - | - |
| 17:16 | INOVER | 0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high | RW | 0x0 |
| 15:14 | Reserved. | - | - | - |
| 13:12 | OEOVER | 0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output | RW | 0x0 |
| 11:10 | Reserved. | - | - | - |
| 9:8 | OUTOVER | 0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high | RW | 0x0 |
| 7:5 | Reserved. | - | - | - |
| 4:0 | FUNCSEL | Function select. 31 == NULL. See GPIO function table for available functions. | RW | 0x1f |



IO Bank Input

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. |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIOX_CTRL: u32 = 0x4001_4004;
5 const GPIO_OE_CLR: *mut u32 = 0xd000_0028 as *mut u32;
6 const GPIO_IN: *const u32 = 0xd000_0004 as *const u32;
7
8 let gpio_ctrl = (GPIOX_CTRL + 8 * pin) as *mut u32;
9
10 let value = unsafe {
11     write_volatile(gpio_ctrl, 5);
12     write_volatile(GPIO_OE_CLR, 1 << pin);
13     read_volatile(GPIO_IN) >> pin & 0b1
14 };
```

GPIOx_CTRL

Offset: 0x004, 0x00c, ... 0x0ec ($0x4 + 8 * x$)

| Bits | Name | Description | Type | Reset |
|-------|-----------|--|------|-------|
| 31:30 | Reserved. | - | - | - |
| 29:28 | IRQOVER | 0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high | RW | 0x0 |
| 27:18 | Reserved. | - | - | - |
| 17:16 | INOVER | 0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high | RW | 0x0 |
| 15:14 | Reserved. | - | - | - |
| 13:12 | OEOVER | 0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output | RW | 0x0 |
| 11:10 | Reserved. | - | - | - |
| 9:8 | OUTOVER | 0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high | RW | 0x0 |
| 7:5 | Reserved. | - | - | - |
| 4:0 | FUNCSEL | Function select. 31 == NULL. See GPIO function table for available functions. | RW | 0x1f |



IO Bank Output

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. |

```
1 use core::ptr::read_volatile;
2 use core::ptr::write_volatile;
3
4 const GPIOX_CTRL: u32 = 0x4001_4004;
5 const GPIO_OE_SET: *mut u32 = 0xd000_0024 as *mut u32;
6 const GPIO_OUT_SET: *mut u32 = 0xd000_0014 as *mut u32;
7 const GPIO_OUT_CLR: *mut u32 = 0xd000_0018 as *mut u32;
8
9 let gpio_ctrl = (GPIOX_CTRL + 8 * pin) as *mut u32;
10 unsafe {
11     write_volatile(gpio_ctrl, 5);
12     write_volatile(GPIO_OE_SET, 1 << pin);
13     let reg = match value {
14         0 => GPIO_OUT_CLR,
15         _ => GPIO_OUT_SET
16     };
17     write_volatile(reg, 1 << pin);
18 };
```

GPIOx_CTRL

Offset: 0x004, 0x00c, ... 0x0ec ($0x4 + 8 * x$)

| Bits | Name | Description | Type | Reset |
|-------|-----------|--|------|-------|
| 31:30 | Reserved. | - | - | - |
| 29:28 | IRQOVER | 0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high | RW | 0x0 |
| 27:18 | Reserved. | - | - | - |
| 17:16 | INOVER | 0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high | RW | 0x0 |
| 15:14 | Reserved. | - | - | - |
| 13:12 | OEOVER | 0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output | RW | 0x0 |
| 11:10 | Reserved. | - | - | - |
| 9:8 | OUTOVER | 0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high | RW | 0x0 |
| 7:5 | Reserved. | - | - | - |
| 4:0 | FUNCSEL | Function select. 31 == NULL. See GPIO function table for available functions. | RW | 0x1f |



Pad Control

GPIOx Register

Offset: 0x004, 0x008, ... 0x078 ($0x4 + 4 \cdot x$)



The User Bank Pad Control registers start at a base address of `0x4001c000` (defined as `PADS_BANK0_BASE` in SDK).

| Offset | Name | Info |
|--------|----------------|----------------------------------|
| 0x00 | VOLTAGE_SELECT | Voltage select. Per bank control |
| 0x04 | GPIO0 | Pad control register |
| 0x08 | GPIO1 | Pad control register |
| 0x0c | GPIO2 | Pad control register |
| 0x10 | GPIO3 | Pad control register |
| 0x14 | GPIO4 | Pad control register |
| 0x18 | GPIO5 | Pad control register |

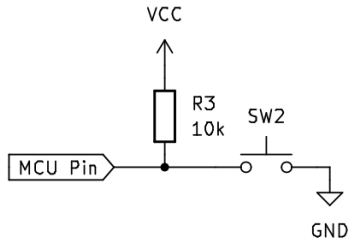
| Bits | Name | Description | Type | Reset |
|------|-----------|--|------|-------|
| 31:8 | Reserved. | - | - | - |
| 7 | OD | Output disable. Has priority over output enable from peripherals | RW | 0x0 |
| 6 | IE | Input enable | RW | 0x1 |
| 5:4 | DRIVE | Drive strength. 0x0 → 2mA 0x1 → 4mA 0x2 → 8mA 0x3 → 12mA | RW | 0x1 |
| 3 | PUE | Pull up enable | RW | 0x0 |
| 2 | PDE | Pull down enable | RW | 0x1 |
| 1 | SCHMITT | Enable schmitt trigger | RW | 0x1 |
| 0 | SLEWFAST | Slew rate control. 1 = Fast, 0 = Slow | RW | 0x0 |



Input

read the value from pin x

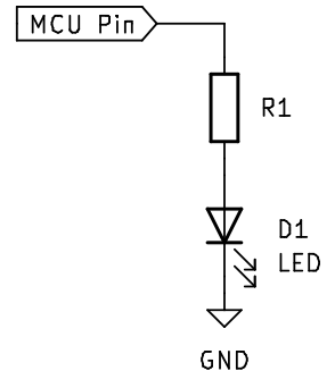
- set the `FUNCSEL` field of `GPIOx_CTRL` to 5
- set the `GPIO_OE_CLR` bit x to 1
- read the `GPIO_IN` bit x
- *adjust the `GPIOx` fields to set the pull up/down resistor*



Output

write a value to pin x

- set the `FUNCSEL` field of `GPIOx_CTRL` to 5
- set the `GPIO_OE_SET` bit x to 1
- if the value
 - is 0, set the `GPIO_OUT_CLR` bit x to 1
 - is 1, set the `GPIO_OUT_SET` bit x to 1
- *adjust the `GPIOx` fields to set the output current*





Rust Embedded HAL

The Rust API for embedded systems



The Rust Embedded Stack

| | |
|---------------------------|--|
| Framework | Tasks, Memory Management, Network etc. <code>embassy-rs</code> , <code>rtic</code> |
| BSC | Board Support Crate <code>embassy-rp</code> , <code>rp-pico</code> |
| <i>HAL Implementation</i> | Uses the PAC and exports a standard HAL towards the upper levels <code>embassy-rp</code> |
| PAC | Accesses registers, usually created automatically from SVD files - <code>rp2040_pac</code> , <code>rp-pac</code> |





GPIO HAL

A set of standard traits

All devices should implement these traits for GPIO.

```
1 pub enum PinState {
2     Low,
3     High,
4 }
```

Input

```
pub trait InputPin: ErrorType {
    // Required methods
    fn is_high(&mut self) -> Result<bool, Self::Error>;
    fn is_low(&mut self) -> Result<bool, Self::Error>;
}
```

Output

```
pub trait OutputPin: ErrorType {
    // Required methods
    fn set_low(&mut self) -> Result<(), Self::Error>;
    fn set_high(&mut self) -> Result<(), Self::Error>;

    // Provided method
    fn set_state(&mut self, state: PinState) -> Result<(), Self::Error>;
}
```




Bare metal

This is how a Rust application would look like

```
1  #![no_std]
2  #![no_main]
3
4  use cortex_m_rt::entry;
5
6  #[entry]
7  fn main() -> ! {
8      // your code goes here
9
10     loop { }
11 }
12
13 #[panic_handler]
14 pub fn panic(_info: &PanicInfo) -> ! {
15     loop { }
16 }
```

Rules

1. never exit the `main` function
2. add a panic handler that does not exit



Bare metal without PAC & HAL

This is how a Rust application would look like

```
1  #![no_std]
2  #![no_main]
3
4  use core::ptr::{read_volatile, write_volatile};
5  use cortex_m_rt::entry;
6
7  const GPIOX_CTRL: u32 = 0x4001_4004;
8  const GPIO_OE_SET: *mut u32 = 0xd000_0024 as *mut u32;
9  const GPIO_OUT_SET: *mut u32 = 0xd000_0014 as *mut u32;
10 const GPIO_OUT_CLR: *mut u32 = 0xd000_0018 as *mut u32;
11
12 #[panic_handler]
13 pub fn panic(_info: &PanicInfo) -> ! {
14     loop { }
15 }
```

```
18  #[entry]
19  fn main() -> ! {
20     let gpio_ctrl = GPIOX_CTRL + 8 * pin as *mut u32;
21     unsafe {
22         write_volatile(gpio_ctrl, 5);
23         write_volatile(GPIO_OE_SET, 1 << pin);
24         let reg = match value {
25             0 => GPIO_OUT_CLR,
26             _ => GPIO_OUT_SET
27         };
28         write_volatile(reg, 1 << pin);
29     };
30
31     loop { }
32 }
```



embassy-rs

Embedded Asynchronous



embassy-rs

- framework
- uses the rust-embedded-hal
- Features
 - Real-time
 - Low power
 - Networking
 - Bluetooth
 - USB
 - Bootloader and DFU



GPIO Input

```
1  #![no_std]
2  #![no_main]
3
4  use embassy_executor::Spawner;
5  use embassy_rp::gpio;
6  use gpio::{Input, Pull};
7
8  #[embassy_executor::main]
9  async fn main(_spawner: Spawner) {
10     let p = embassy_rp::init(Default::default());
11     let pin = Input::new(p.PIN_3, Pull::Up);
12
13     if pin.is_high() {
14
15     } else {
16
17     }
18 }
```

The `main` function is called by the embassy-rs framework, so it can exit.



GPIO Output

```
1  #![no_std]
2  #![no_main]
3
4  use embassy_executor::Spawner;
5  use embassy_rp::gpio;
6  use gpio::{Level, Output};
7
8  #[embassy_executor::main]
9  async fn main(_spawner: Spawner) {
10     let p = embassy_rp::init(Default::default());
11     let mut pin = Output::new(p.PIN_2, Level::Low);
12
13     pin.set_high();
14 }
```

The `main` function is called by the embassy-rs framework, so it can exit.



Conclusion

we discussed about

- Memory Mapped IO
- RP2040 GPIO
 - Single Cycle IO
 - IO Bank
 - Pad
- The Rust embedded standard stack
- Bare metal Rust
- The embassy-rs framework