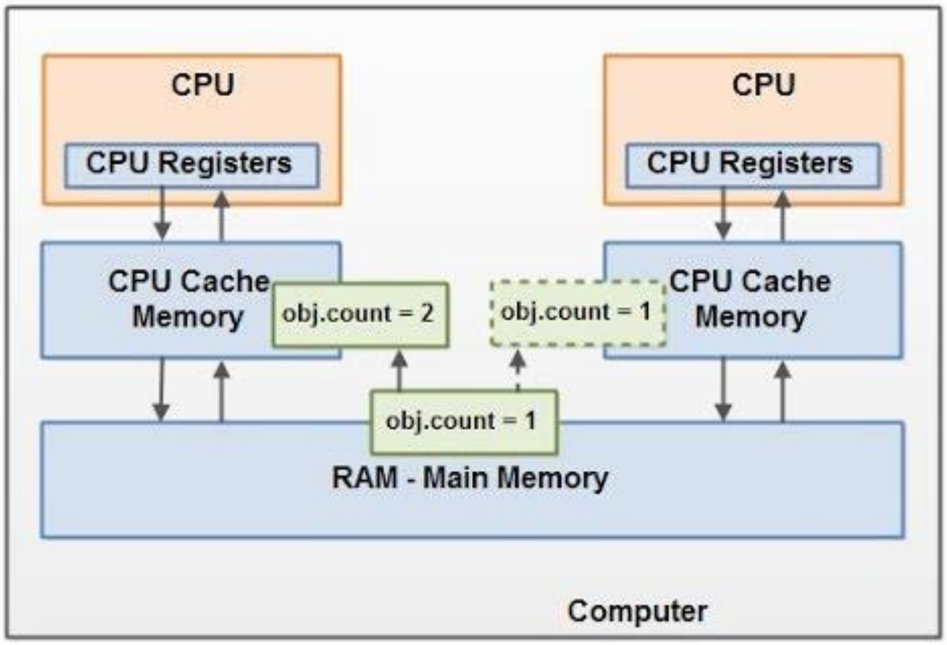
Chapter 7

Registers

Registers

- Registers are type of computer memory
- They are reside near to CPU and used immediately by CPU
- Can hold data (bits of sequence), storage address (pointers) or instructions.
- Registers holding the memory location are used to calculate the address of the next instruction.
- In this chapter we will revisit the LED’s program, but this time we will see things in a bit depth
- In terms of MCU’s registers are special memory regions

Registers



CPU
CPU Registers
CPU Cache Memory
obj.count = 2
CPU
CPU Registers
CPU Cache Memory
obj.count = 1
obj.count = 1
RAM - Main Memory
Computer

Registers

- These special regions map to or control some peripherals
- And we already know we have many of of them in our board
- These peripherals are electronic components built in the MCU to provide it's processor the ability to interact with IO devices.
- We will be following code from this repository

https://github.com/PIAIC-IOT/Quarter2-Online.git

Registers

- Here is the starter code:

```rust
#[allow(unused_imports)]
use aux::{entry, iprint, iprintln};

#[entry]
fn main() {
    aux::init();

    unsafe {

        const GPIOE_BSRR: u32 = 0x48001018;

        (GPIOE_BSRR as *mut u32) = 1 << 9;
        (GPIOE_BSRR as *mut u32) = 1 << 11;
        (GPIOE_BSRR as *mut u32) = 1 << (9 + 16);
        (GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
    }
    loop {}
}
```

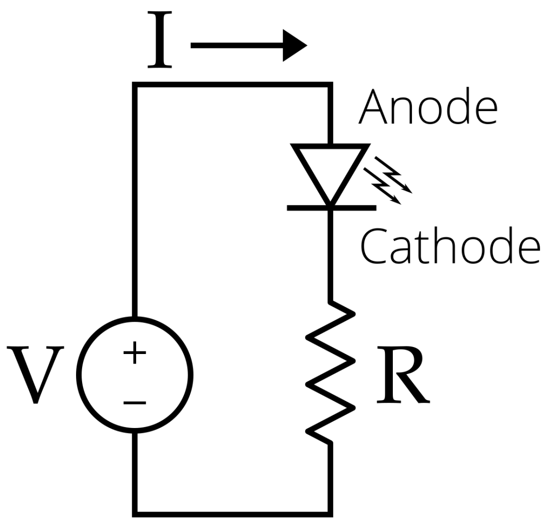
Registers

- We are using raw pointer here for demonstration purpose. It's not recommended to use in normal programs btw except in genuine case
- Address (i.e. **0x48001018**) points to a register
- This register controls **GPIO** pins. **GPIO** is a peripheral
- Also using GPIO we can drive the mapped pins to low or high

Light Emitting Diode

Light Emitting Diode (LED)

- Pins are electrical contacts and our MCU has many of them
- Few of them are connected to LEDs, few are connected to other devices on board
- Once a pin connected to an LED in right polarity and a high voltage applied to that pin that's the only case LED will glow



Reading The Reference Manual

RTRM

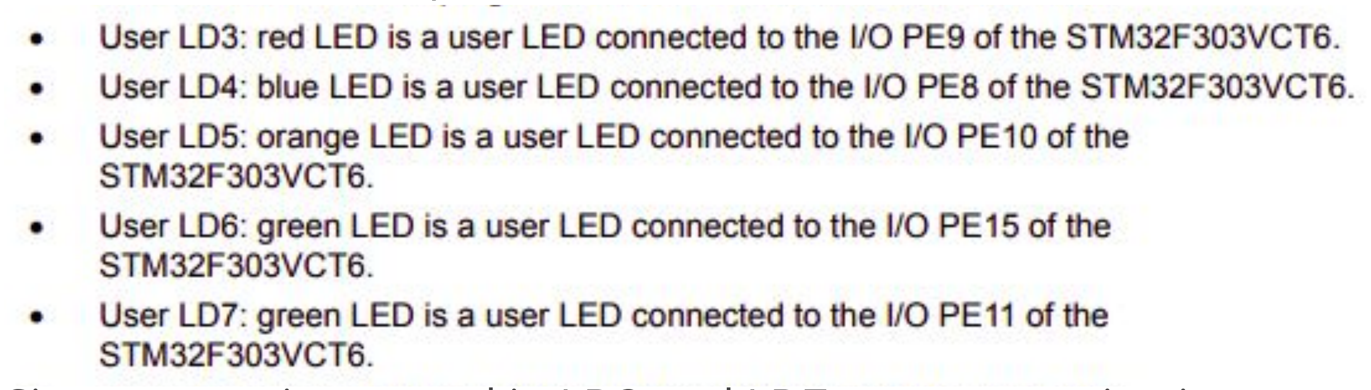
Whenever we start learning a new programming language, using a new hardware or buy a new cell phone. It's documentation/guide is **must read/follow** thing to take full advantage of that.

- Manual says that MCU has several pins.
- These pins are **grouped in ports** of 16 pins
- Each port named with a letter (e.g. A, B, C, .. , H)
- Pins within each port named with a number (e.g. 0, 1, 2, .. , 15)

Since we are interested in finding pins associated with LEDs, so lets check the manual to find out those pins.

RTRM

On page 18, under section 6.4 (LEDs) it is mention which LED connected to which pin.



- User LD3: red LED is a user LED connected to the I/O PE9 of the STM32F303VCT6.
- User LD4: blue LED is a user LED connected to the I/O PE8 of the STM32F303VCT6.
- User LD5: orange LED is a user LED connected to the I/O PE10 of the STM32F303VCT6.
- User LD6: green LED is a user LED connected to the I/O PE15 of the STM32F303VCT6.
- User LD7: green LED is a user LED connected to the I/O PE11 of the STM32F303VCT6.

Since we are interested in LD3 and LD7 as we are using in our program.

RTRM

- Manual says LD3 connected to PE9 and LD7 connected to PE11
- Here "PE" in PEXX represents the port name (i.e. Port E of GPIO peripheral)
- The number after the port name represents pin number in the port.

Conclusion:

After this exercise we find out what pins we need to **low**/**high** to **on**/**off** LEDs.

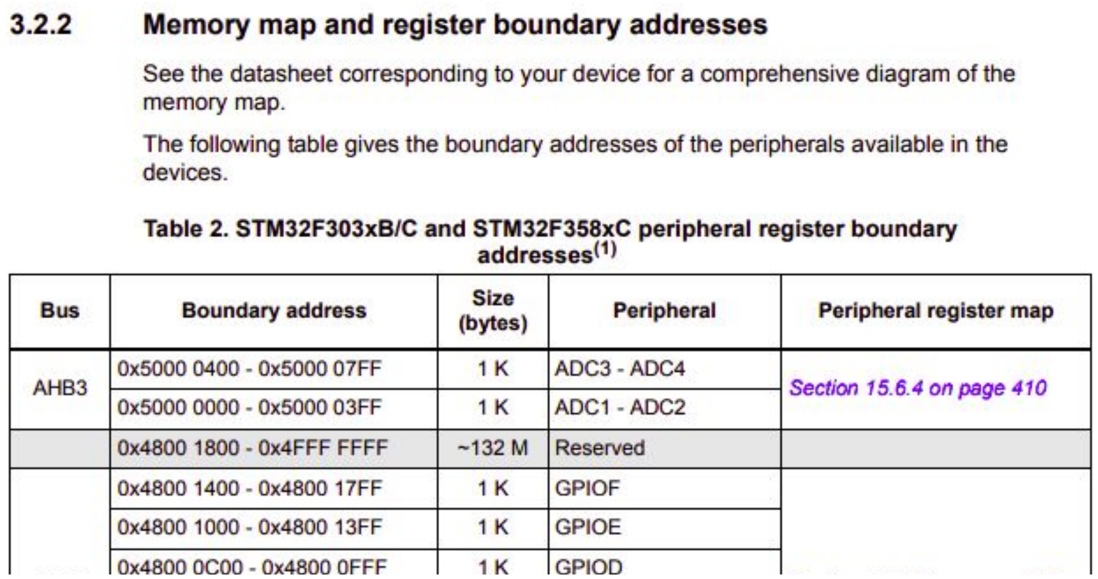
RTRM

- Each peripheral has a register block associated to it.
- A register block is a **collection of registers** allocated in **memory contiguously**
- The starting address of this block called **base address**
- Now we know to driving LEDs we need to manipulate with Port E registers. For that we need to find out the base address of Port E.

For finding base address of Port E we have to follow the reference manual

RTRM

- On page 51, under section 3.2.2 (Memory map and register boundary addresses) boundary addresses of Port E mentioned.



3.2.2 Memory map and register boundary addresses

See the datasheet corresponding to your device for a comprehensive diagram of the memory map.

The following table gives the boundary addresses of the peripherals available in the devices.

Table 2. STM32F303xB/C and STM32F358xC peripheral register boundary addresses[1]

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
AHB3	0x5000 0400 - 0x5000 07FF	1 K	ADC3 - ADC4	*Section 15.6.4 on page 410*
	0x5000 0000 - 0x5000 03FF	1 K	ADC1 - ADC2	
	0x4800 1800 - 0x4FFF FFFF	~132 M	Reserved	
	0x4800 1400 - 0x4800 17FF	1 K	GPIOF	
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE	
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD	

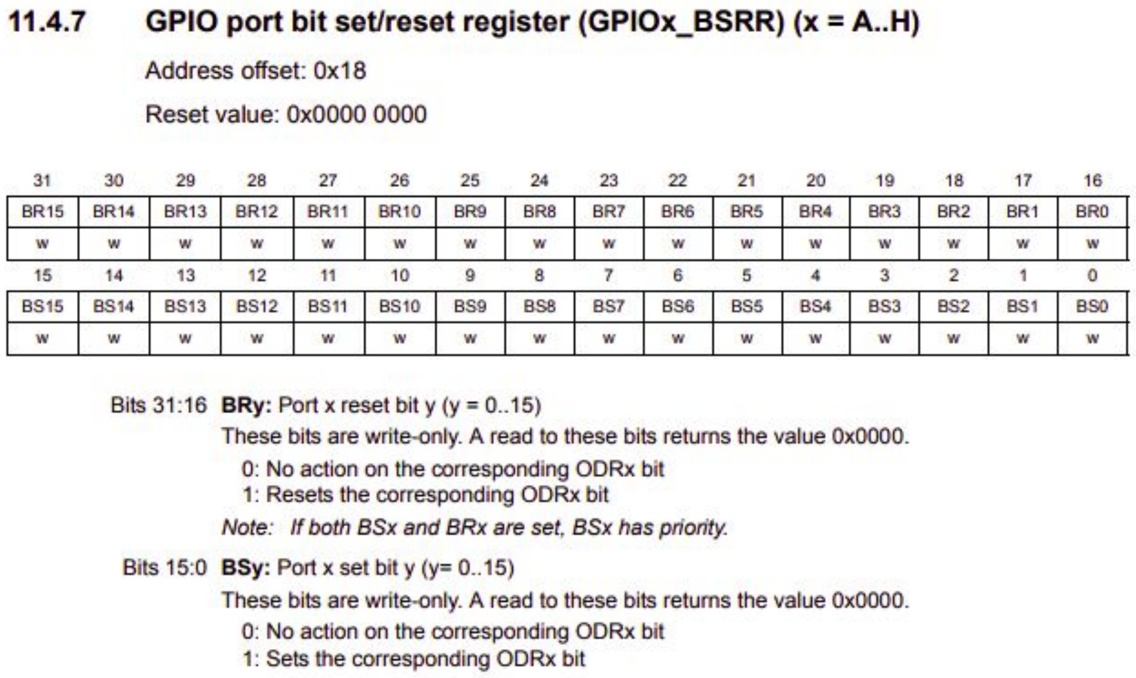
RTRM

At this point we found the base address of Port E (0x48001000)

- Now we need to target a register that will let us set/reset the bits of Port E.
- That register is BSRR => Bit Set Reset Register.
- For even that we need to consult manual

Section 11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) - Page 240

RTRM



11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

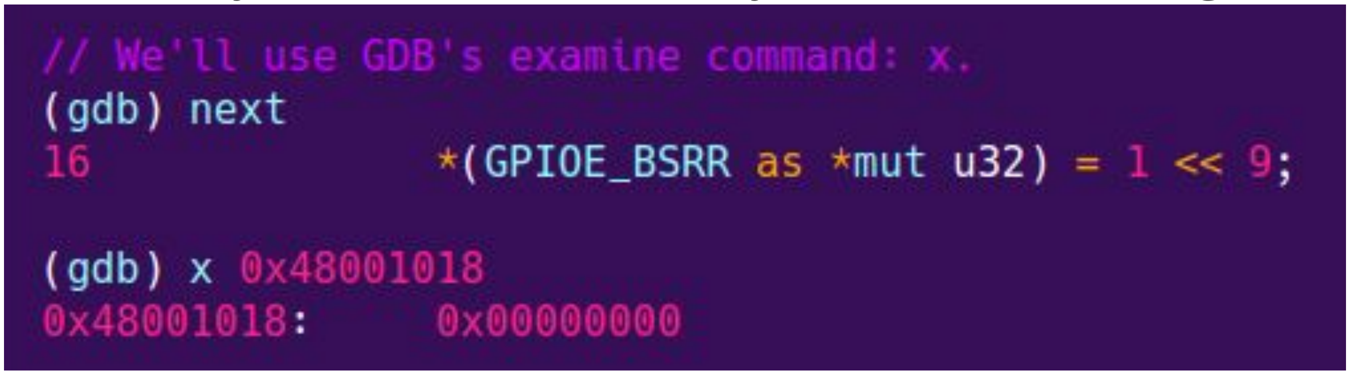
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

RTRM

- Documentation says it is **write-only** register, which means we can write to this register only but can't read from it.
- To verify this we will code and try to read from this register



```
// We'll use GDB's examine command: x.
(gdb) next
16              *(GPIOE_BSRR as *mut u32) = 1 << 9;

(gdb) x 0x48001018
0x48001018:     0x00000000
```

- Reading the register return us **0**

RTRM

Few other informations we extracted from the documentation:

- Bits **0-15** can be used to set the corresponding pins (i.e. bit 0 sets pin 0 and bit 15 sets bit 15)
- Also, bits **16-31** can be used to reset the corresponding pins

Correlating that information with our program, all seems to be in agreement:

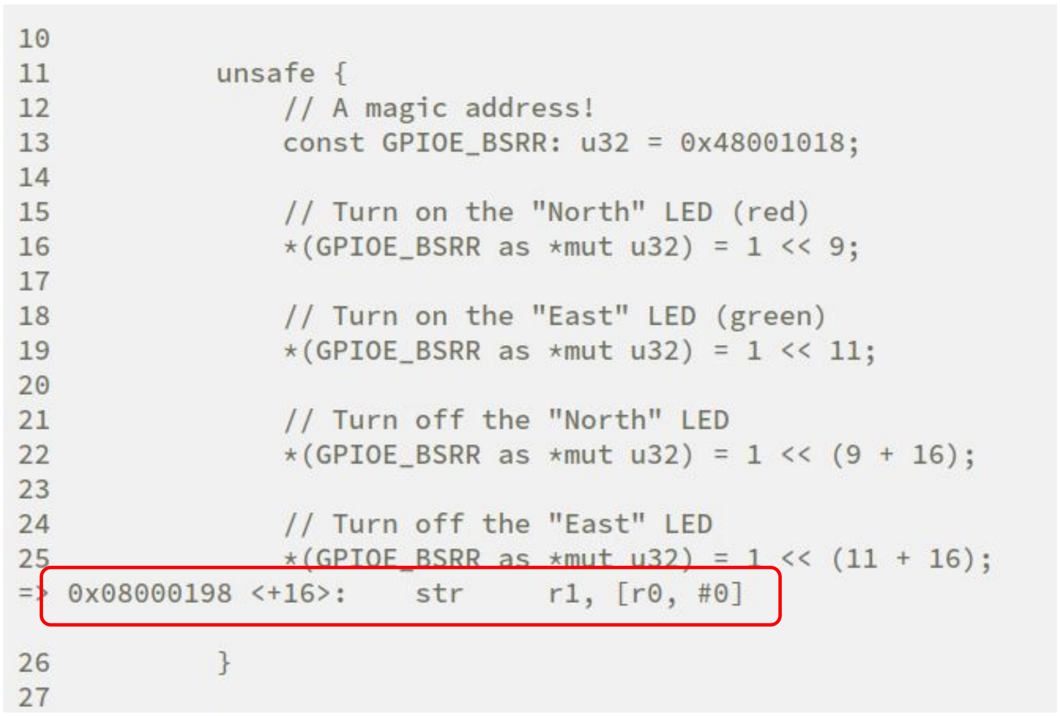
- Writing **1 << 9** (BS9 = 1) to BSRR sets PE9 high
- Writing **1 << 11** (BS11 = 1) to BSRR sets PE11 high
- Writing **1 << 25** (BR9 = 1) to BSRR sets PE9 low
- Finally, writing **1 << 27** (BR11 = 1) to BSRR sets PE11 low

mis(Optimization)

mis(Optimization)

- Read/write to registers are special operations
- Have some side-effects
- We wrote 4 different values to the same address
- If not knowing upfront we end up writing only **1 << (11 + 16)**
- LLVM doesn't know that we're writing to register
- It optimizes the code and will merge all the writes
- For verifying this we can run **cargo run --release**
- After running the only line in the code we run this command **disassemble /m**

mis(Optimization)



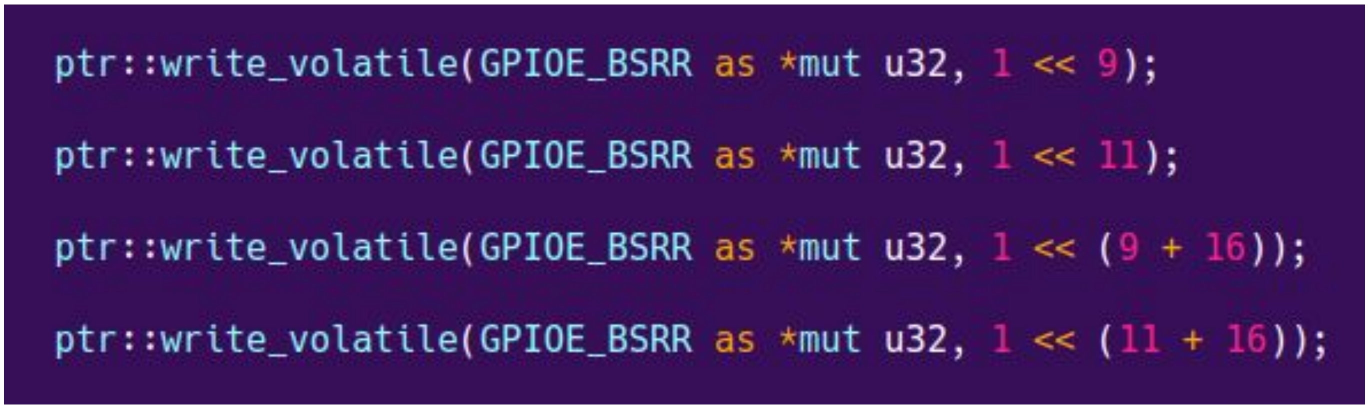
```

        unsafe {
            // A magic address!
            const GPIOE_BSRR: u32 = 0x48001018;

            // Turn on the "North" LED (red)
            *(GPIOE_BSRR as *mut u32) = 1 << 9;

            // Turn on the "East" LED (green)
            *(GPIOE_BSRR as *mut u32) = 1 << 11;

            // Turn off the "North" LED
            *(GPIOE_BSRR as *mut u32) = 1 << (9 + 16);

            // Turn off the "East" LED
            *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
=> 0x08000198 <+16>:    str     r1, [r0, #0]

        }

```

mis(Optimization)

- We will observe this time LED won’t change its state
- The noted **str** instruction is the one which writes the values to the register
- Our debug (unOptimized) version has four of them
- To verify this we will run this command **cargo objdump --bin registers -- -d -no-show-raw-insn -print-imm-hex -source**

Solution

- There is a solution for this mis(Optimized) problem
- That is **volatile operation** rather than plain read/write operation
- This will prevent LLVM from optimizing our program
- Below is the code we need to substitute in our program

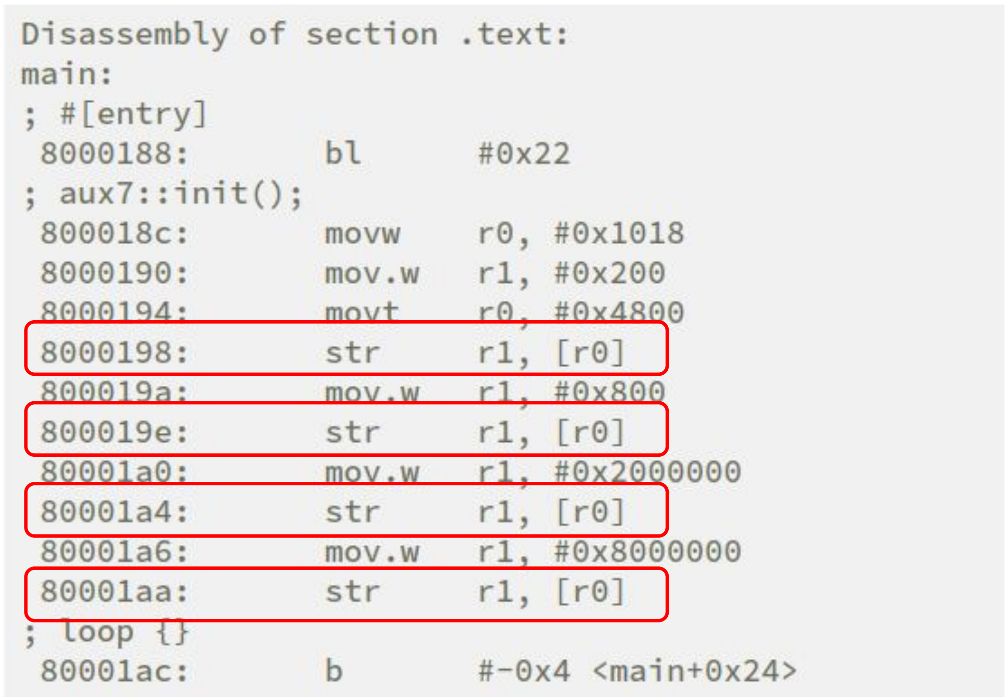


```rust
ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 9);

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 11);

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (9 + 16));

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (11 + 16));
```

Solution

- Now if we run the following command, we will observe that our program has 4 **str** instructions in optimized version.
- **cargo objdump --bin registers --release -- -d -no-show-raw-insn -print-imm-hex -source**

Solution

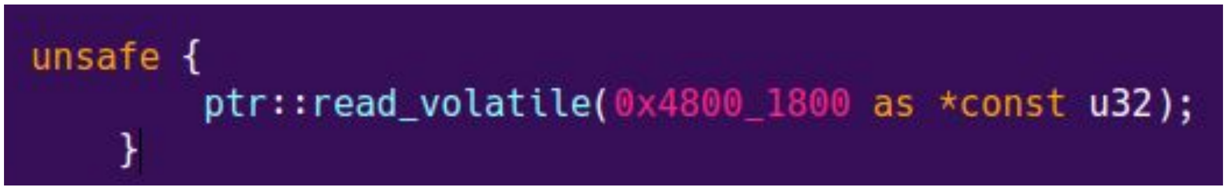


```
Disassembly of section .text:
main:
; #[entry]
 8000188:       bl      #0x22
; aux7::init();
 800018c:       movw    r0, #0x1018
 8000190:       mov.w   r1, #0x200
 8000194:       movt    r0, #0x4800
 8000198:       str     r1, [r0]
 800019a:       mov.w   r1, #0x800
 800019e:       str     r1, [r0]
 80001a0:       mov.w   r1, #0x2000000
 80001a4:       str     r1, [r0]
 80001a6:       mov.w   r1, #0x8000000
 80001aa:       str     r1, [r0]
; loop {}
 80001ac:       b       #-0x4 <main+0x24>
```

0xBAAAAAAD Address

0xBAAAAAAD Address

- Not all the memory addresses can be accessed
- Look at the code below, in this if we try to access this address



```
unsafe {
        ptr::read_volatile(0x4800_1800 as *const u32);
    }
```

- This address is close to GPIOE_BSRR register's address but this one is invalid
- There is no register at this address
- If we run this code, we will get the following error

0xBAAAAAAD Address

```
$ cargo run
Breakpoint 3, main () at src/07-registers/src/main.rs:9
9           aux7::init();

(gdb) continue
Continuing.

Breakpoint 2, UserHardFault_ (ef=0x10001fc0)
    at $REGISTRY/cortex-m-rt-0.6.3/src/lib.rs:535
535         loop {
```

- This exception raised because we tried to access an address that doesn't exist.

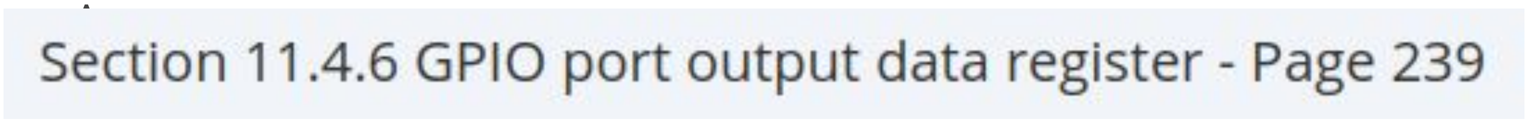
Exceptions

- Exceptions are raised when processor tries to perform an invalid operation
- Exception breaks the normal flow of our program
- It forces the processor to execute an exception handler
- Exception handler is just a function/subroutine
- There different kind of exceptions
- Each one raised by different condition
- Raised exception even handled by different exception handler

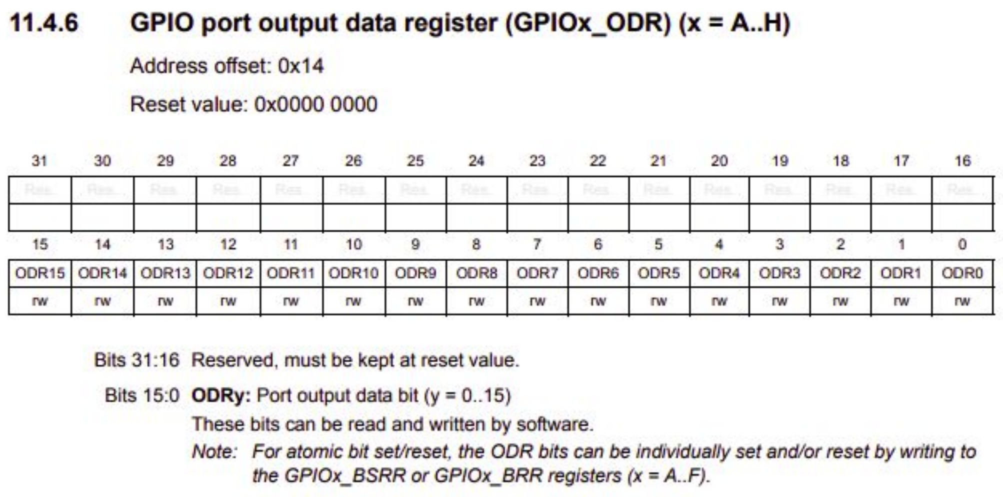
Spooky action at a distance

Spooky action at a distance

- **BSRR** is not the only register that can control the pins
- **ODR** is the register that give you read/write access to the port's pins
- For **ODR** register details, we will consult to the manual again


Section 11.4.6 GPIO port output data register - Page 239

Spooky action at a distance



11.4.6 GPIO port output data register (GPIOx_ODR) (x = A..H)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

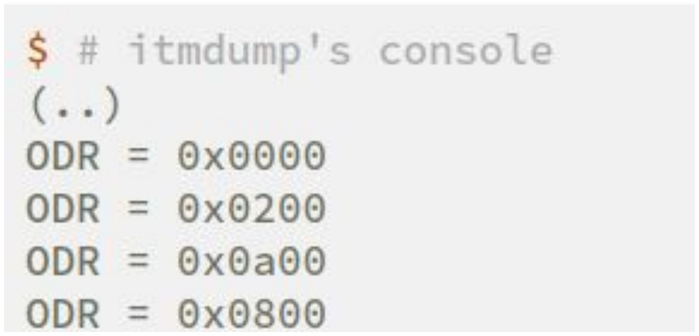
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16	Reserved, must be kept at reset value.
Bits 15:0	**ODRy:** Port output data bit (y = 0..15) These bits can be read and written by software. *Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).*

- Lets try running the code below in the description, which writes to BSRR and reads from ODR register.

Spooky action at a distance

We will be getting the following result on the **itmdump console**

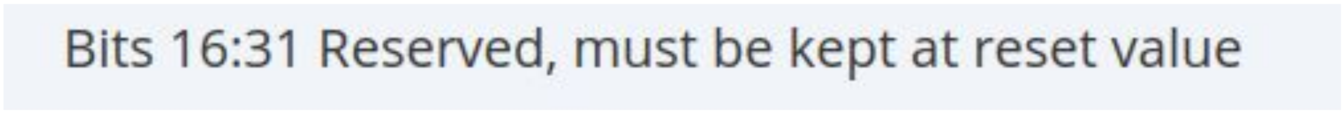


```
$ # itmdump's console
(..)
ODR = 0x0000
ODR = 0x0200
ODR = 0x0a00
ODR = 0x0800
```

Type safe manipulation

Type safe manipulation

- Last register we were working with was ODR register, and documentation says:



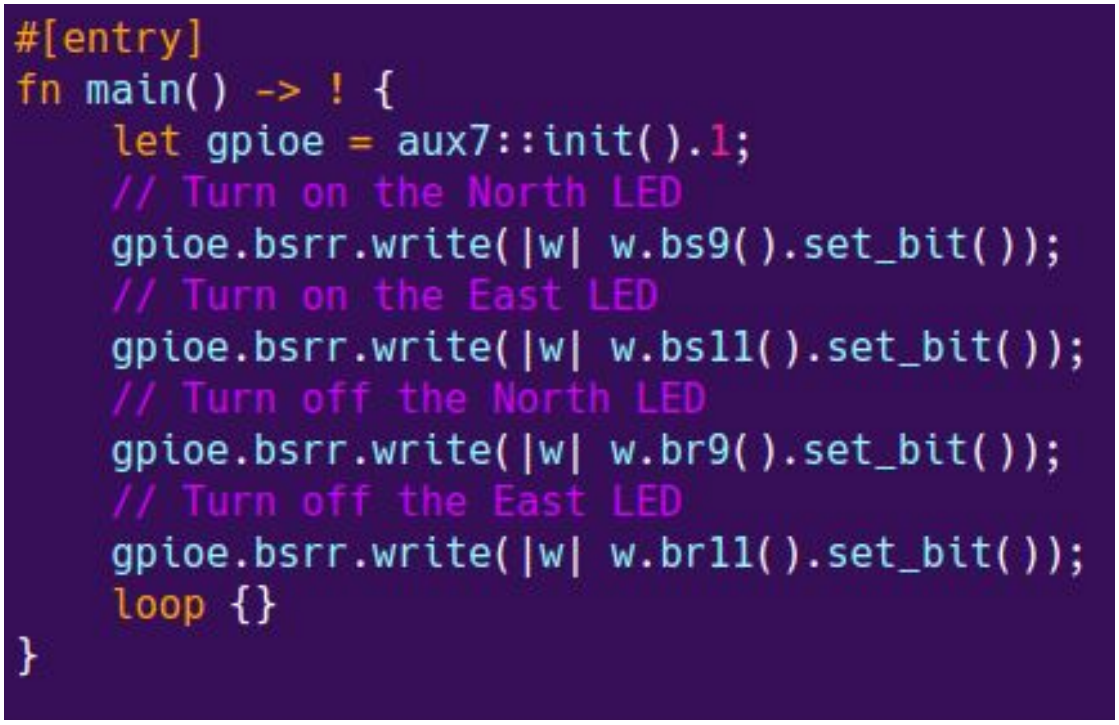
Bits 16:31 Reserved, must be kept at reset value

- So, we can not write to these bits otherwise something unexpected may happen
- Peripheral ports in MCU have many registers and each one has different read/write permissions
- Directly working with hexadecimal addresses is error prone

API Usage

- Unintentionally, if tried to access an invalid memory location it will break our program
- So wouldn't it be nice if we write code in safe manner using API
- The API will then take care of the concerns we have while directly interacting with registers
- We won't be dealing with addresses directly
- We won't have to worry about read/write permissions
- We also won't be hitting the reserved area of registers
- Now lets see how we do it

API Usage



```
#[entry]
fn main() -> ! {
    let gpioe = aux7::init().1;
    // Turn on the North LED
    gpioe.bsrr.write(|w| w.bs9().set_bit());
    // Turn on the East LED
    gpioe.bsrr.write(|w| w.bs11().set_bit());
    // Turn off the North LED
    gpioe.bsrr.write(|w| w.br9().set_bit());
    // Turn off the East LED
    gpioe.bsrr.write(|w| w.br11().set_bit());
    loop {}
}
```

API Usage

- If you observed there is no magic address now
- Also it is more human readable way to access BSRR using **gpioe.bsrr**
- Then we are having **write** closure that responsible for setting/resetting the bits of register
- Another thing to notice that we have 2 methods **bsx()** and **brx()** to set the values

Summary