Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

CSE 3105

Computer Interfacing & Embedded System

Memory Mapped I/O

Md. Nasif Osman Khansur
Lecturer
Dept. of CSE, RUET

References

1. Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C 3e by Dr. Yifeng Zhu [Chapter 14]

I/O Operations

- Typically, an on-chip peripheral device has a few registers, such as control registers, status registers, data input registers, and data output registers. A peripheral may also, have data buffers, such as the display memory of LCD controller. Input/output or I/O refers to data communication between the processor core and a peripheral device.
- There are two complementary approaches to performing I/O operations: port-mapped, and memory-mapped l/O.

A **Peripheral** is a hardware device with a specific address in memory that it writes data to and/or reads data from.

I/O Operations

Port-mapped I/O:

- Port-mapped I/O uses special machine instructions, which are designed specifically for l/O operations.
- The memory address space and the l/O device address space are independent of each other.
- Each device is assigned one or more address space unique port numbers. For example, Intel x86 processors use IN OUT instructions to read from or write to a port.

I/O Operations

Memory-mapped I/O:

- Memory-mapped I/O does not need any special instructions. The memory and I/O devices share the same address space.
- Each peripheral register or data buffer is assigned to a memory address in the memory address space of the microprocessor.
- Memory-mapped I/O is performed by the native load and store instructions of the processor. Therefore, memory-mapped l/O is a more convenient way to interface I/O devices.
- The most significant disadvantage is that memory-mapped l/O has a more complex address decoding unit than port-mapped l/O.

I/O Operations

Memory Mapped IO or MMIO is the process of interacting with hardware devices by reading from and writing to predefined memory addresses.

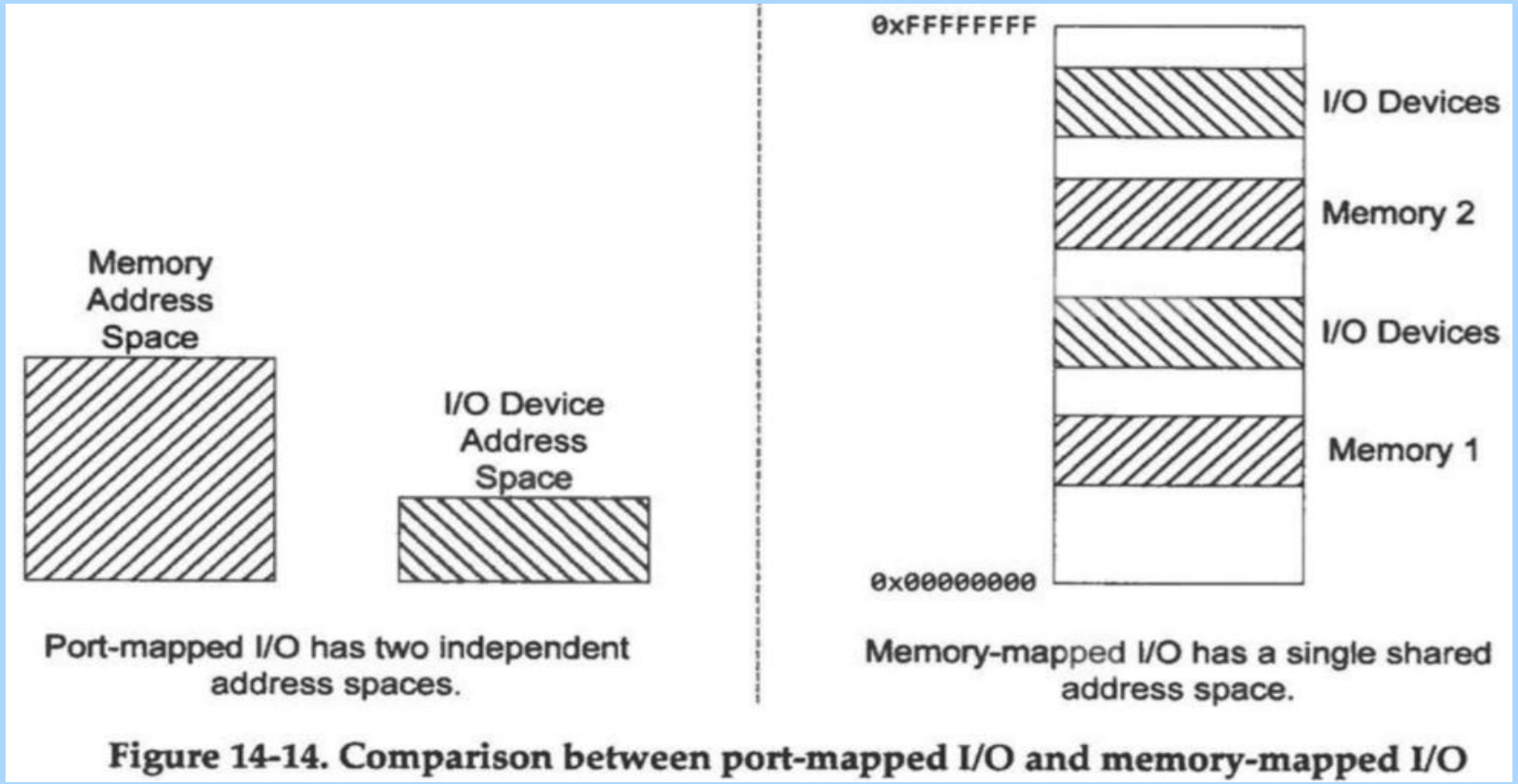
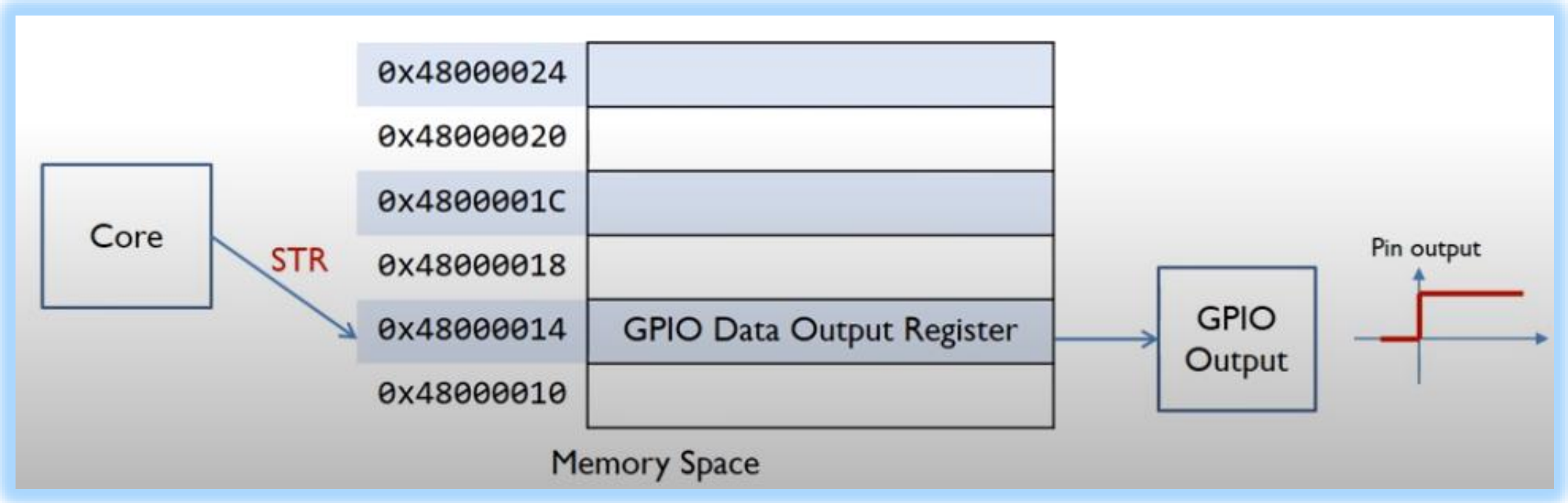


Figure 14-14. Comparison between port-mapped I/O and memory-mapped I/O

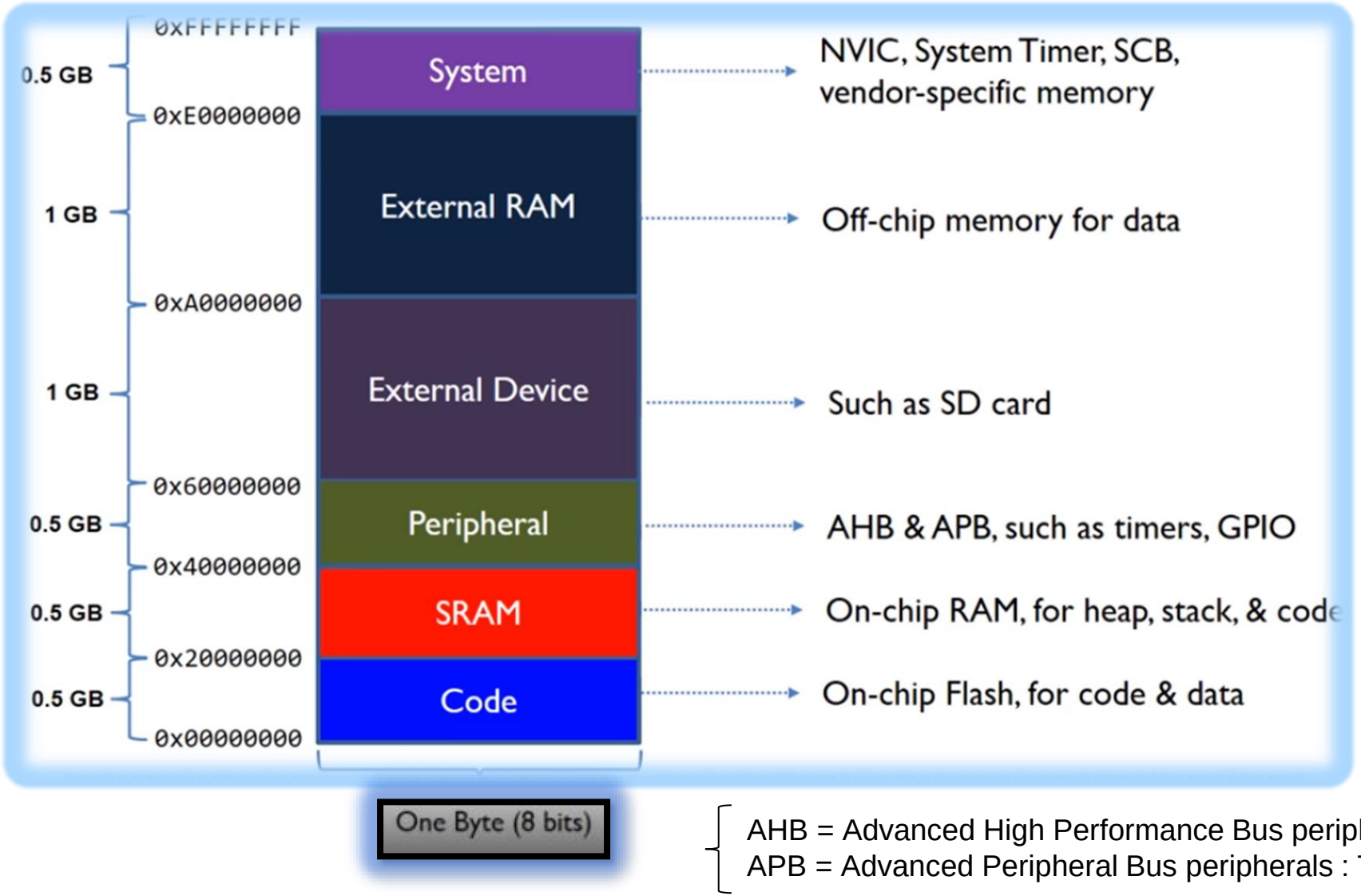
MMIO


0x48000024
0x48000020
0x4800001C
0x48000018
0x48000014
0x48000010
GPIO Data Output Register
Core
STR
GPIO Output
Pin output
Memory Space

I/O Operations

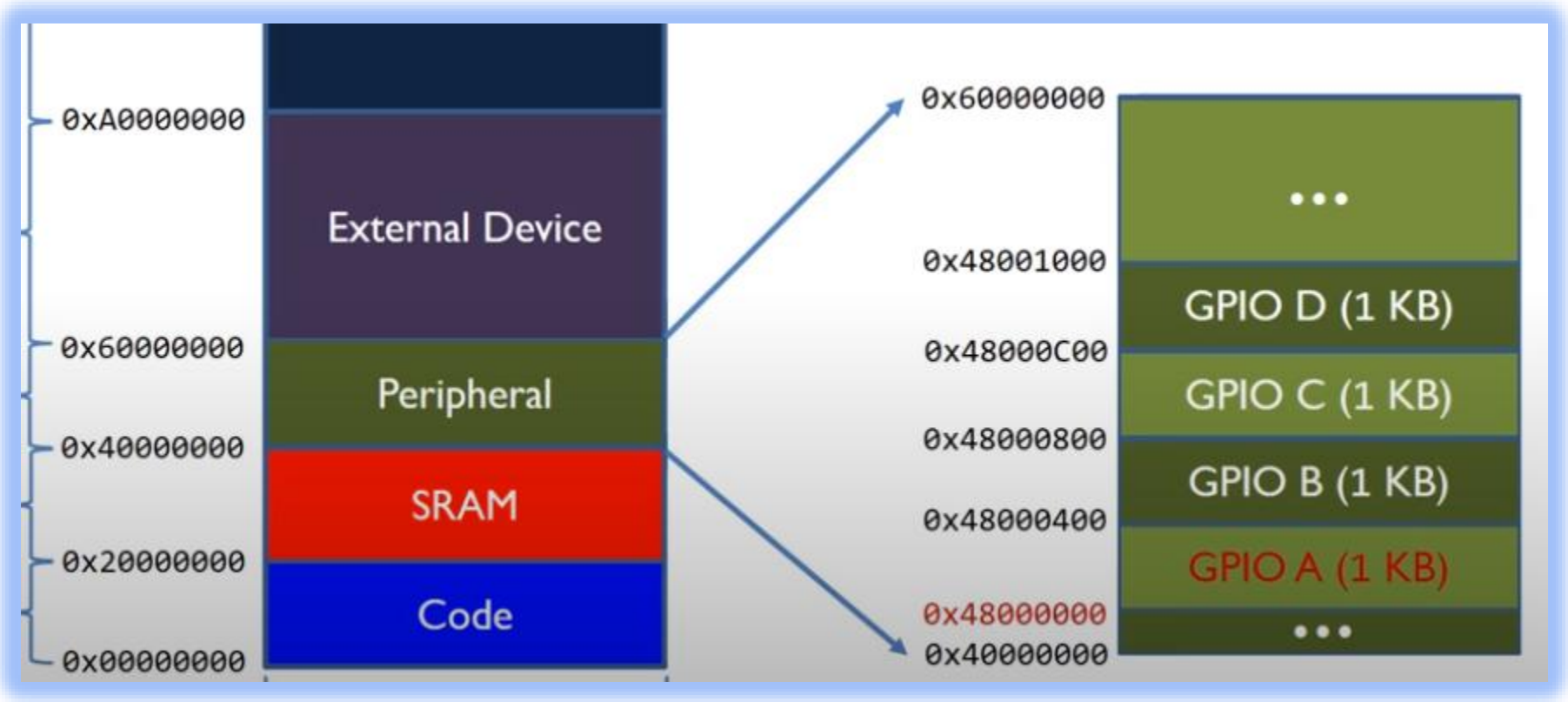
- ARM Cortex-M processors use memory-mapped I/O to access peripheral registers. All peripheral registers on STM32L4 are mapped to a small memory region starting at 0x40000000.
- This region includes the memory addresses of all on-chip peripherals, such as GPIO, timers, UART, SPI, and ADC. The memory address of each peripheral register is determined by chip manufacturers, and usually cannot be changed by software.
- A peripheral register usually takes four bytes in memory.
  - For example, the output data (ODR) register of Port B on STM32L4 is mapped to memory addresses 0x48000414 to 0x48000417, with the upper halfword being reserved. Note that values stored in peripheral registers are in the format of little-endian.

I/O Operations



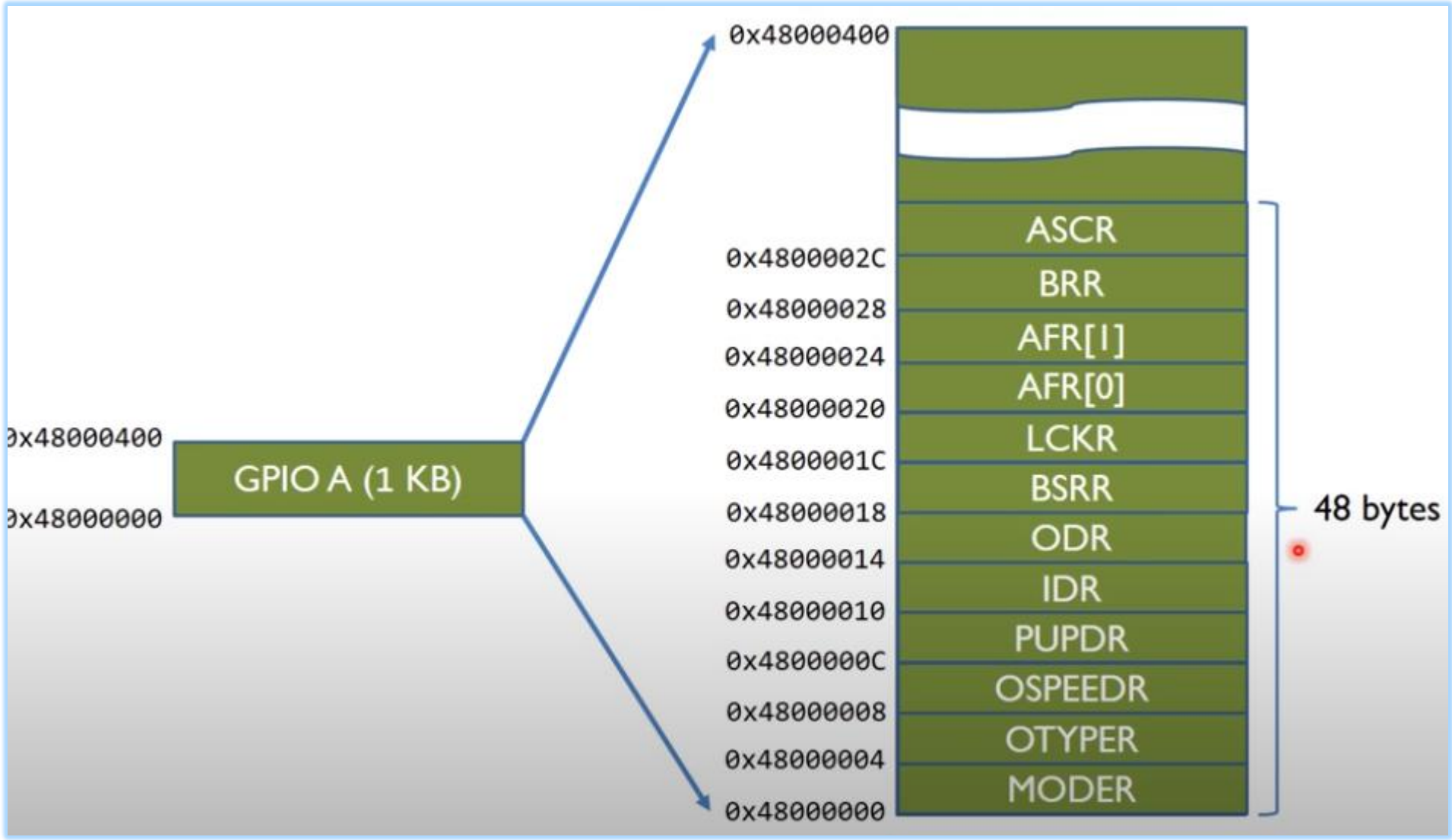
AHB = Advanced High Performance Bus peripheral : GPIO, ADC
APB = Advanced Peripheral Bus peripherals : Timers, USART

I/O Operations



On STM32L4, the registers of GPIO Port A, are mapped to a small memory region starting at 48000000 in hex.

I/O Operations

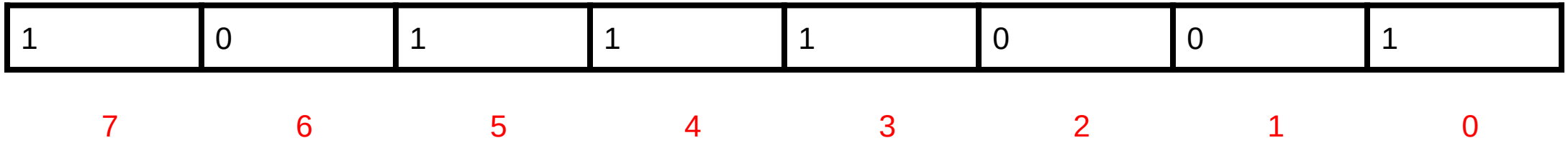


Properties of GPIO Registers

- The memory addresses of the registers are defined during the chip design stage. Software cannot change them.
- Each GPIO port has up to 16 pins, and each pin may take 1, 2, or 4 bits in a control register. For example, two bits are required to specify the mode of a GPIO pin. Therefore, the size of these control registers can be either 2, 4 or 8 bytes.
- Because the memory address of each register is word-aligned (is a multiple of four), dummy bytes are padded in the data structure to correctly map the fixed physical memory layout to the data structure.

Some Basics

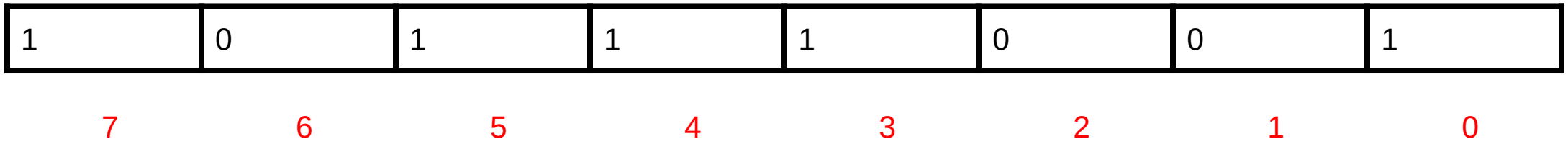
Suppose we are working on a 8-bit processor. The ODR (Output Data Register) looks like this.



What if we want to change the bit at position 6 and 5 to 10?

Some Basics

Suppose we are working on a 8-bit processor. The ODR (Output Data Register) looks like this.



What if we want to change the bit at position 6 and 5 to 10?

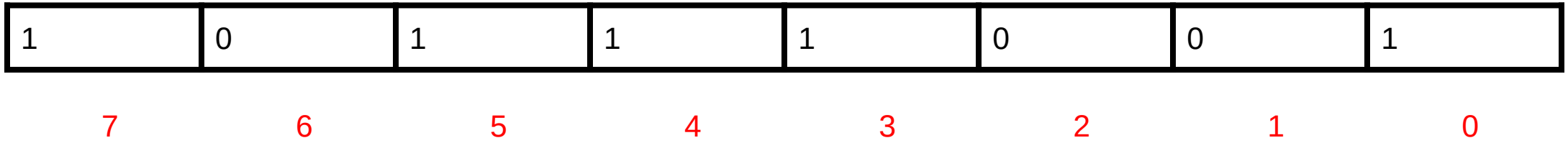
1. Erase the bits – Perform an AND operation between ODR and 10011111.

10011111 → ~01100000 → (bin)~11<<5 → (dec)~3<<5

These summarizes to this– ODR &= ~3<<5

Some Basics

Suppose we are working on a 8-bit processor. The ODR (Output Data Register) looks like this.



What if we want to change the bit at position 6 and 5 to 10?

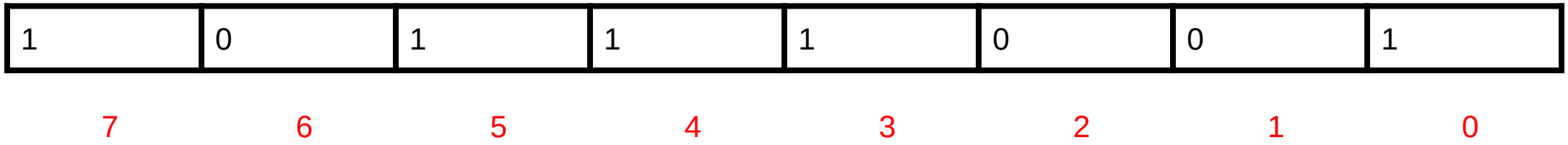
1. Erase the bits – Perform an AND operation between ODR and 10011111.

10011111 → ~01100000 → (bin)~11<<5 → (dec)~3<<5

These summarizes to this– ODR &= ~3<<5 [ODR → 10011001]

Some Basics

Suppose we are working on a 8-bit processor. The ODR (Output Data Register) looks like this.



What if we want to change the bit at position 6 and 5 to 10?

1. Erase the bits – Perform an AND operation between ODR and 10011111.

10011111 → ~01100000 → (bin)~11<<5 → (dec)~3<<5

These summarizes to this– ODR &= ~3<<5 [ODR → 10011001]

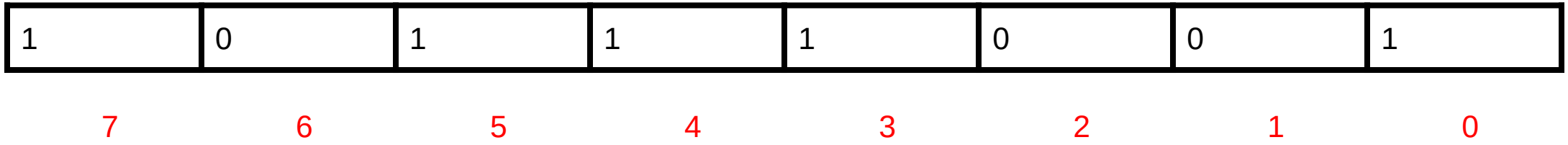
1. Change the bits – Perform an OR operation between ODR and 01000000.

01000000 → (bin)10<<5 → (dec)2<<5

These summarizes to this– ODR |= 2<<5

Some Basics

Suppose we are working on a 8-bit processor. The ODR (Output Data Register) looks like this.



What if we want to change the bit at position 6 and 5 to 10?

1. Erase the bits – Perform an AND operation between ODR and 10011111.

10011111 → ~01100000 → (bin)~11<<5 → (dec)~3<<5

These summarizes to this– ODR &= ~3<<5 [ODR → 10011001]

1. Change the bits – Perform an OR operation between ODR and 01000000.

01000000 → (bin)10<<5 → (dec)2<<5

These summarizes to this– ODR |= 2<<5 [ODR → 11011001]

Some Basics

Let’s think of a real example. In STM32L4 MCU, the ODR of GPIO port B starts from the memory address 0x48000414 and spans 32 bits. We want to set the register at position 13 (i.e., B13). How do we do that?

1. define a pointer that points to memory address 0x48000414.

Some Basics

Let's think of a real example. In STM32L4 MCU, the ODR of GPIO port B starts from the memory address 0x48000414 and spans 32 bits. We want to set the register at position 13 (i.e., B13). How do we do that?

1. define a pointer that points to memory address 0x48000414.

   #define GPIOB_ODR (volatile uint32_t *) 0x48000414

Typically, compilers minimize the number of memory accesses, by temporally storing the memory value in a register, and then repeatedly using it without accessing the memory. The volatile qualifier on a variable prevents the compiler from making such optimization on this variable.

Some Basics

Let’s think of a real example. In STM32L4 MCU, the ODR of GPIO port B starts from the memory address 0x48000414 and spans 32 bits. We want to set the register at position 13 (i.e., B13). How do we do that?

1. define a pointer that points to memory address 0x48000414.

   #define GPIOB_ODR (volatile uint32_t *) 0x48000414

1. Dereference the address to access the value stored and set the value.

Some Basics

Let's think of a real example. In STM32L4 MCU, the ODR of GPIO port B starts from the memory address 0x48000414 and spans 32 bits. We want to set the register at position 13 (i.e., B13). How do we do that?

1. define a pointer that points to memory address 0x48000414.

   #define GPIOB_ODR (volatile uint32_t *) 0x48000414

1. Dereference the address to access the value stored and set the value.

   *GPIOB_ODR |= 1UL<<13

UL = Unsigned Long Integer

Programming Memory Mapped I/O

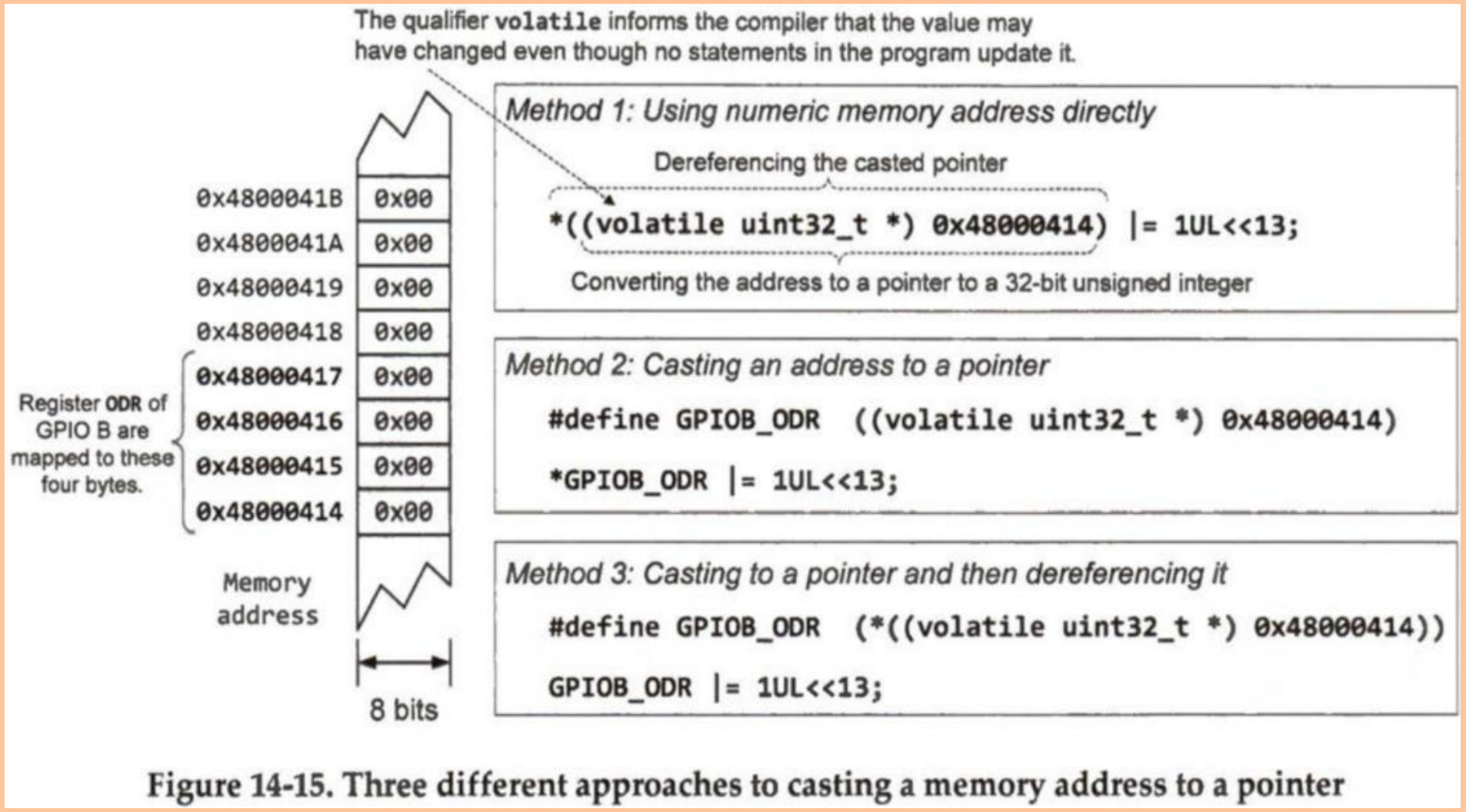


Figure 14-15. Three different approaches to casting a memory address to a pointer

Programming Memory Mapped I/O

- A better approach is to use structures and pointers. Typically, all registers of a peripheral, such as those of GPIO port B (shown in Figure 14-16), are mapped to a contiguous block of physical memory. In C, a struct encapsulates related variables into a single structure. All variables in a struct are stored contiguously in memory.

```
#define __IO  volatile  // allows read and write

typedef struct{
  __IO uint32_t MODER;   // Mode register
  __IO uint16_t OTYPER;  // Output type register
       uint16_t rev0;    // Padding two bytes
  __IO uint32_t OSPEEDR; // Output speed register
  __IO uint32_t PUPDR;   // Pull-up/pull-down register
  __IO uint16_t IDR;     // Input data register
       uint16_t rev1;    // Padding two bytes
  __IO uint16_t ODR;     // Output data register
       uint16_t rev2;    // Padding two bytes
  __IO uint16_t BSRRL;   // Bit set/reset register(low)
  __IO uint16_t BSRRH;   // Bit set/reset register(high)
  __IO uint32_t LCKR;    // Configuration lock register
  __IO uint32_t AFR[2];  // Alternate function registers
  __IO uint32_t BRR;     // Bit reset register
  __IO uint32_t ASCR;    // Analog switch control register
} GPIO_TypeDef

#define GPIOB ((GPIO_TypeDef *) 0x48000400))
```

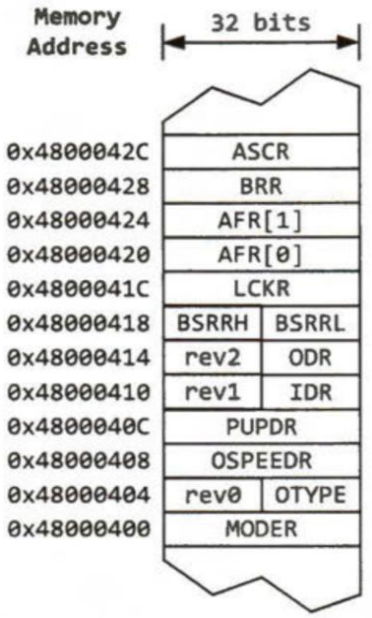


Figure 14-16. Casting the memory address of GPIO B to a GPIO structure pointer.

Programming Memory Mapped I/O

- In STM32L4 MCU, GPIOB registers start from 0x48000400 and spans 48 bytes. The defined GPIO_TypeDef is also 48 bytes. So, when we are pointing GPIOB to 0x48000400, the 48 bytes/the 12 registers of 4-bytes are going to map to the 12 4-bytes blocks of the structure. So, instead of accessing ODR by it's address, we can access it by simply GPIOB -> ODR.

```
#define __IO  volatile  // allows read and write

typedef struct{
  __IO uint32_t MODER;    // Mode register
  __IO uint16_t OTYPER;   // Output type register
       uint16_t rev0;     // Padding two bytes
  __IO uint32_t OSPEEDR;  // Output speed register
  __IO uint32_t PUPDR;    // Pull-up/pull-down register
  __IO uint16_t IDR;      // Input data register
       uint16_t rev1;     // Padding two bytes
  __IO uint16_t ODR;      // Output data register
       uint16_t rev2;     // Padding two bytes
  __IO uint16_t BSRRL;    // Bit set/reset register(low)
  __IO uint16_t BSRRH;    // Bit set/reset register(high)
  __IO uint32_t LCKR;     // Configuration lock register
  __IO uint32_t AFR[2];   // Alternate function registers
  __IO uint32_t BRR;      // Bit reset register
  __IO uint32_t ASCR;     // Analog switch control register
} GPIO_TypeDef

#define GPIOB ((GPIO_TypeDef *) 0x48000400))
```

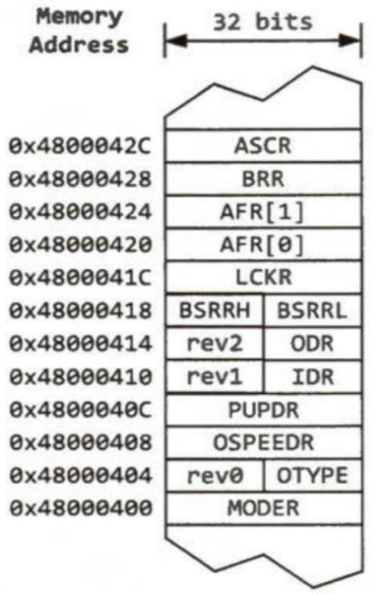


Figure 14-16. Casting the memory address of GPIO B to a GPIO structure pointer.

Programming Memory Mapped I/O

- To conveniently access a set of registers which are contiguous in memory, we can cast the base memory address of a GPIO port to a pointer to a data structure, as shown below.

  #define GPIOB ((GPIO_TypeDef *) 0x48000400))

- For example, if we want to set the output of GPIO port B pin **6** to high, we can use the following C statement. **1UL** is an unsigned long integer with a value of **1**. Note the pins are numbered **0-15**, instead of **1-16**.

  GPIOB->ODR |= 1UL << 6; //Set bit 6

Programming Memory Mapped I/O

In C

```
GPIOA->ODR |= 1UL<<14;    // Set bit 14 to 1
```

In Assembly

```
GPIOA_BASE EQU   0x48000000
GPIO_ODR   EQU   0x14

LDR r7, =GPIOA_BASE        ; Load GPIO port A base address
LDR r1, [r7, #GPIO_ODR]    ; r1 = GPIOA->ODR
ORR r1, r1, #(1<<14)       ; Set output of pin 14 to high
STR r1, [r7, #GPIO_ODR]    ; Write the output data register
```

Thank You!