Introduction to the STM32 L432KC MCU & Refresher on C Programming

Lecture 06

Josh Brake
Harvey Mudd College

Outline

- Introduction to the STM32-L432KC
- Review of basic architecture
- C refresher
  - Common idioms to set/clear bits
  - Pointers and arrays
  - Structures
- Writing a simple device driver: GPIO
  - Finding information in documentation
  - Writing code to properly configure the peripheral

Learning Objectives

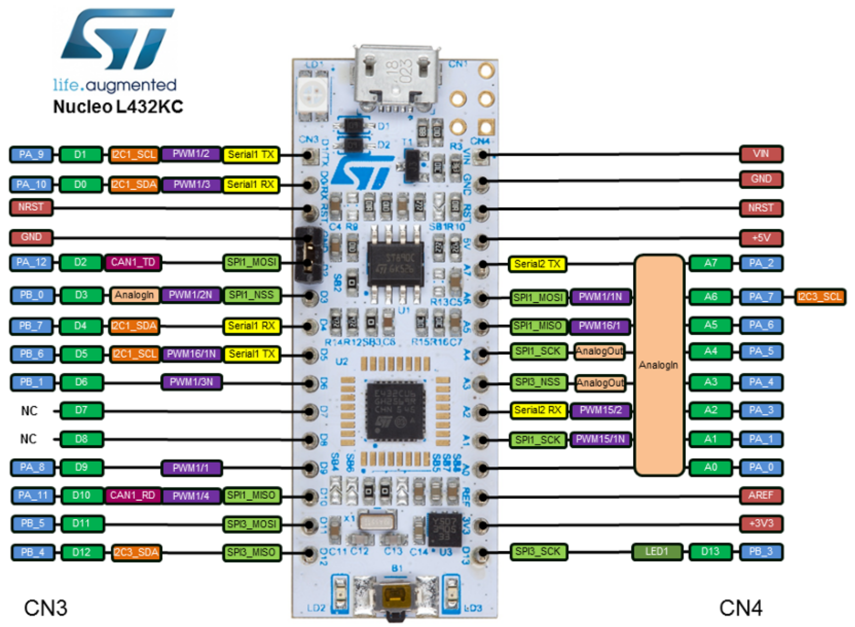
By the end of this lecture you should be able to…

- List the basic details of the architecture of the ARM Cortex-M4 CPU used in our STM32 MCU.
- Recall basic C programming idioms and concepts (e.g., pointers, arrays, structures).
- Write a simple device driver to control the peripherals in your MCU using memory-mapped I/O.

STM32 Nucleo-32 board

Components

- MCU – STM32L432KC
- External flash memory
- 24 MHz crystal oscillator
- On-board ST-LINK debugger/programmer. Virtual COM port and debug port.
- 1 user LED and 1 reset push button
- Arduino Nano V3 form factor



What is an MCU

- MCU = MicroController Unit
- Processor core + peripherals

Figure 1. System architecture

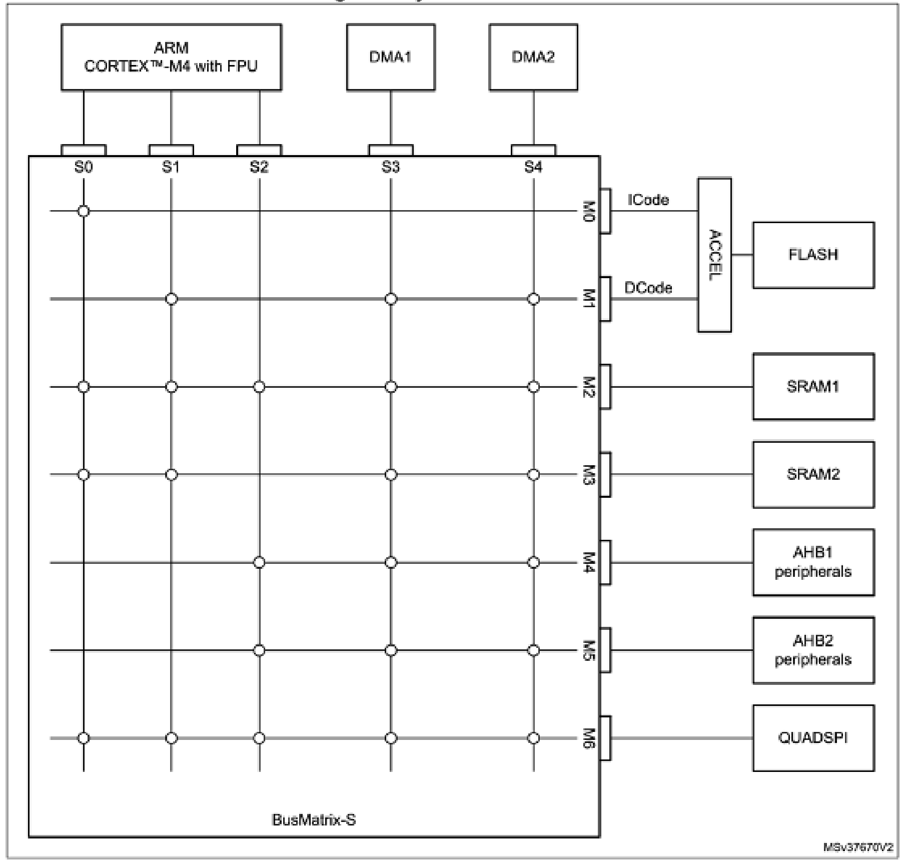
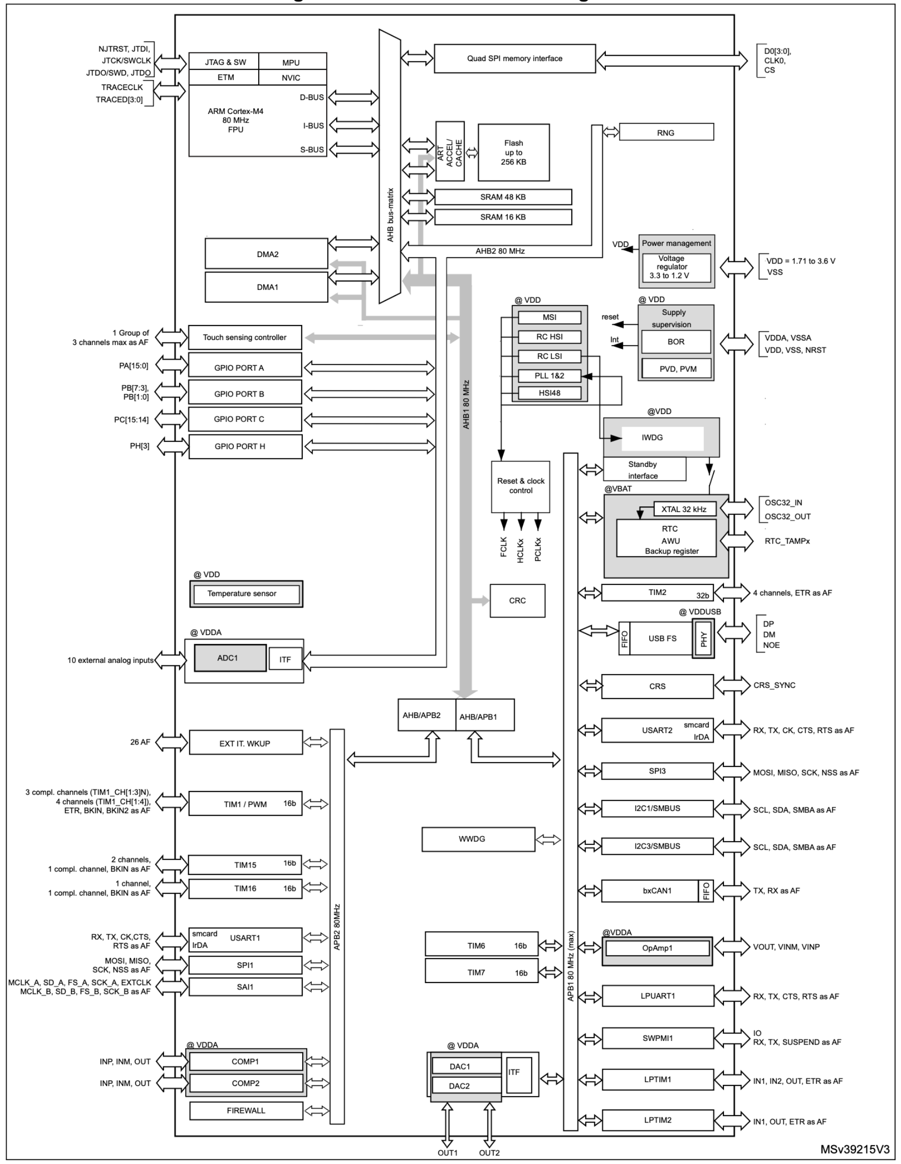


Figure 1. STM32L432xx block diagram



Note: AF: alternate function on I/O pins.

Documentation

Reference Manual: Information about all peripherals and their control registers

Datasheet: System block diagram, Pin functions, electrical characteristics, timing specs

Programmer’s Manual: Information about the architecture (e.g., Cortex-M4), supported assembly instructions, registers, memory map, etc.

Questions for a new MCU

- What is the register set?
- What does the memory map look like?
- What addressing modes are used?
- What types of instructions exist?
- What I/O functions are available?

STM32 L432KC Architecture

- STM32 MCU has an ARM Cortex-M4.
- It runs the ARMv7E-M architecture. This is a 32-bit architecture.
- Also supports Thumb-2 execution. Thumb-2 is a set of compressed, 16-bit instructions.
- One special thing to watch with Thumb is conditional execution. With Thumb execution you can only use conditional execution within an “if-then” block which can hold up to four successive instructions.

Architecture Overview

- Instructions are 32-bit
- Sixteen 32-bit registers R0-R15
- Supports condition codes
- Most instructions operate on two registers and put result in a third.

Microarchitecture Flashback

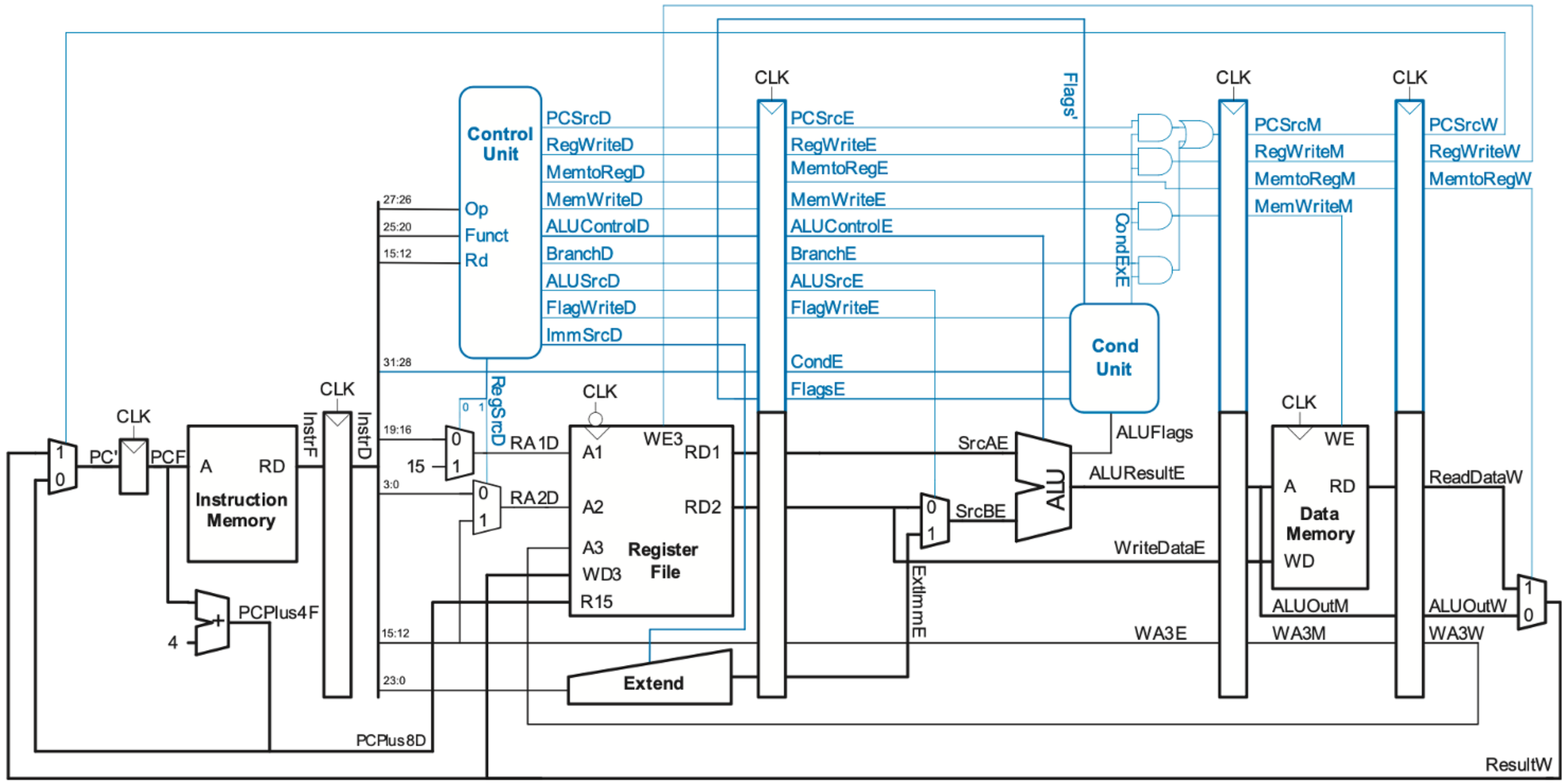


Figure 7.47 **Pipelined processor with control**

Harris and Harris, *Digital Design and Computer Architecture ARM Ed.*, p. 430

Register Set

- R15 is Program Counter (PC)
- R14 is Link Register (LR). Holds return addresses
- R13 by convention used as the stack pointer.
- Four condition codes in current program status register (CPSR)
  - N – negative
  - Z – zero
  - C – carry (unsigned overflow)
  - V – (signed) overflow

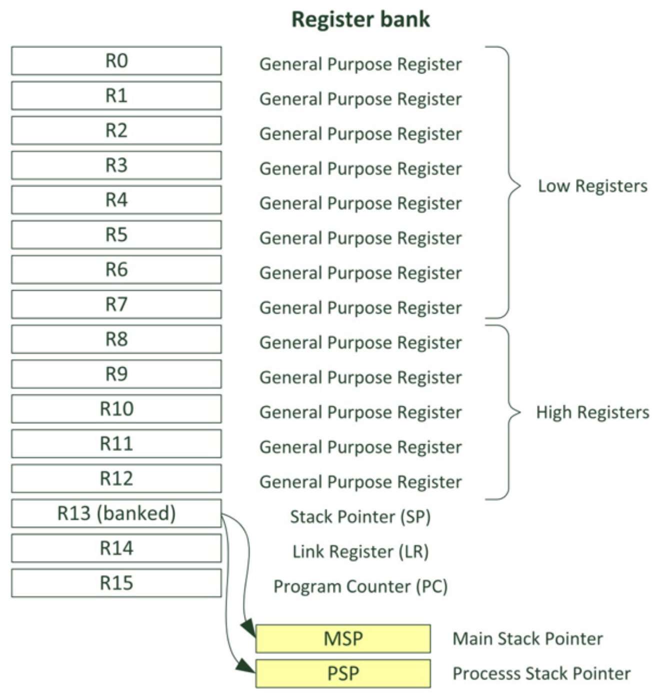


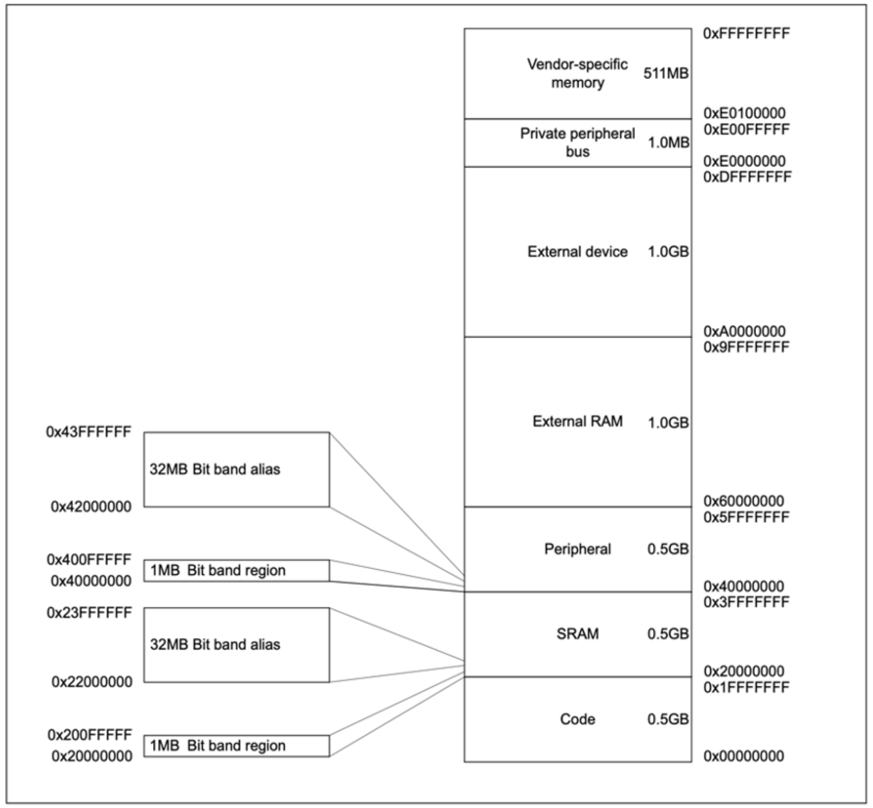
FIGURE 4.3

Registers in the register bank

Memory Map

- Flat 32-bit instruction set
- Addressed in bytes.
- 2^{32} bytes of memory accessible (4 Gigabytes)
- Instructions are always aligned on word (4-byte) in standard ARM and halfword (2-byte) boundaries in Thumb mode.

Figure 8. Memory map

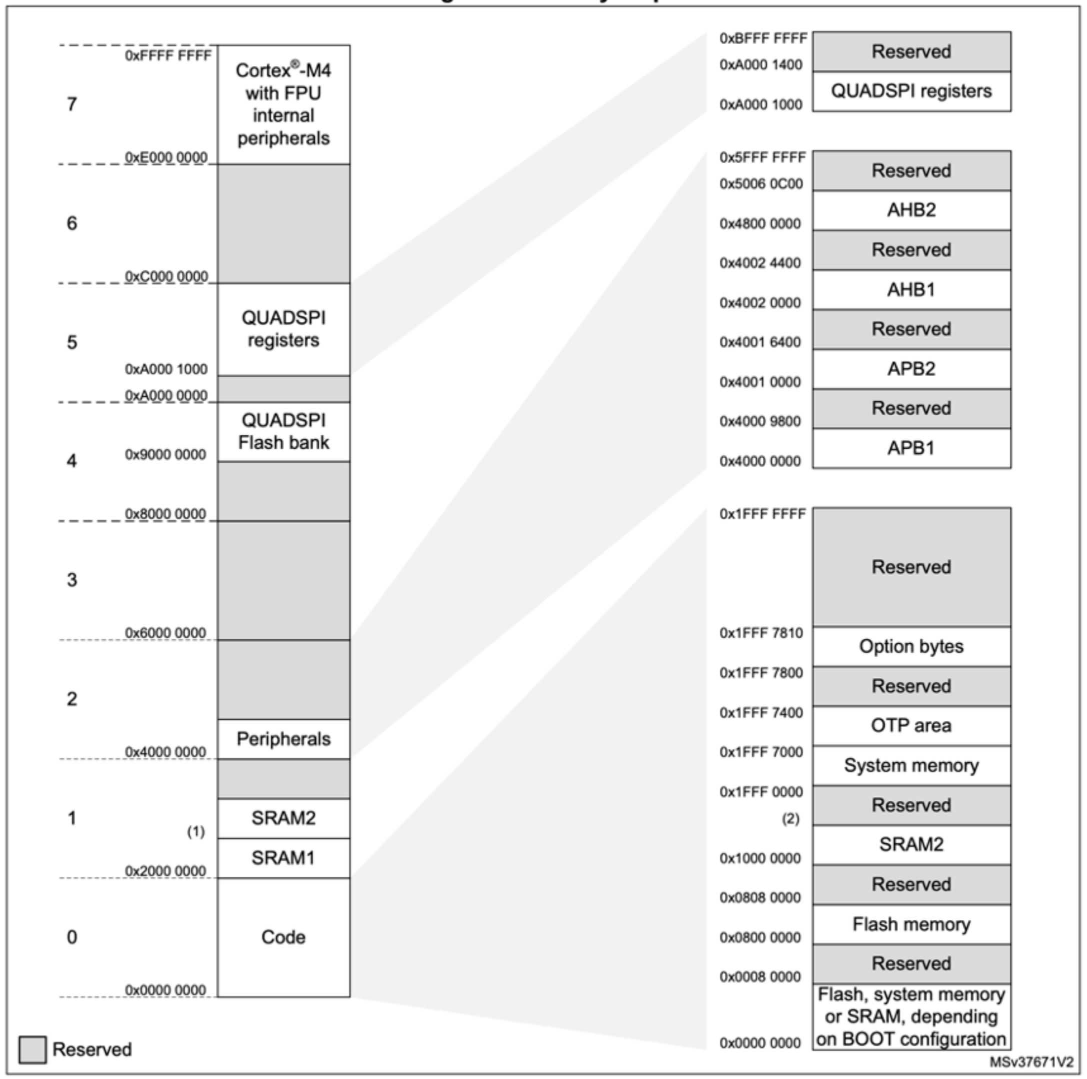


PM0214 Cortex-M4 Programmers Manual

Memory Map - STM32L432KC

RM0394 p. 67

Figure 2. Memory map



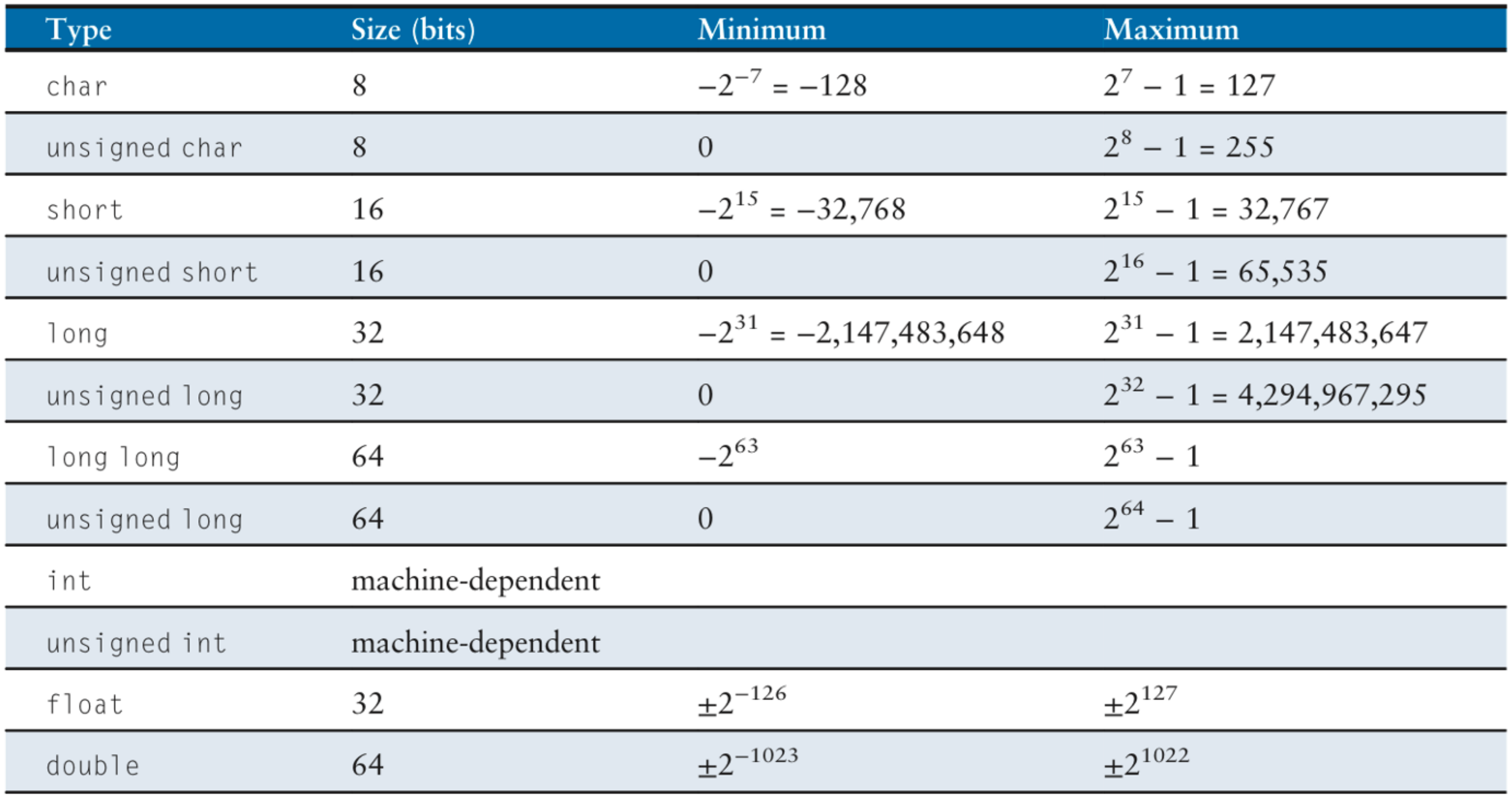
C Programming Review

Important Concepts in C

- C is libertarian by nature. You can stomp on any memory address you want!
- There is no memory management built in. You must manually allocate (and deallocate!) any memory you need.

Primitive Data Types in C

Table eC.2 Primitive data types and sizes



Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^{7} - 1 = 127$
`unsigned char`	8	0	$2^{8} - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Primitive Data Types in `stdint.h`

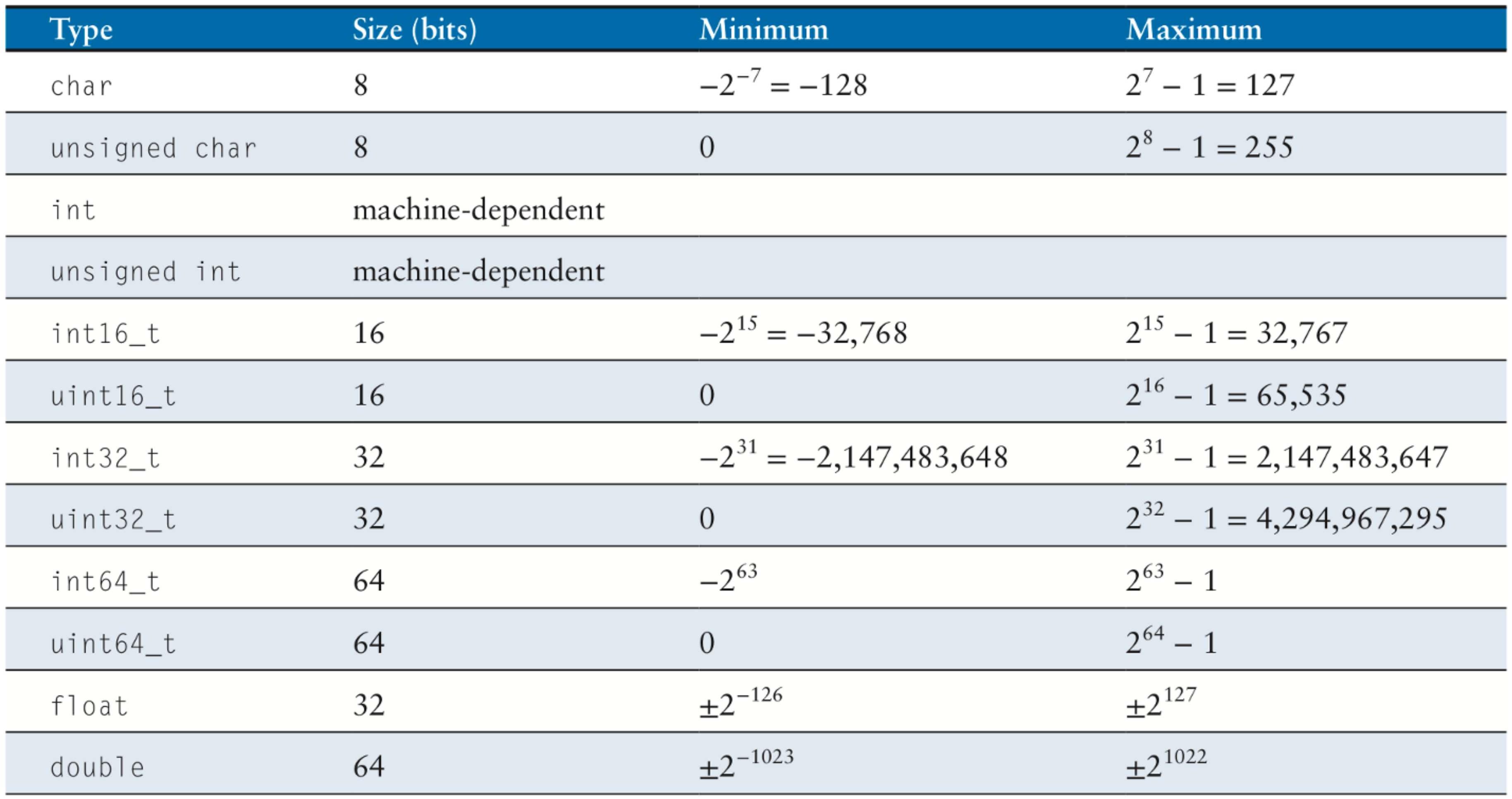
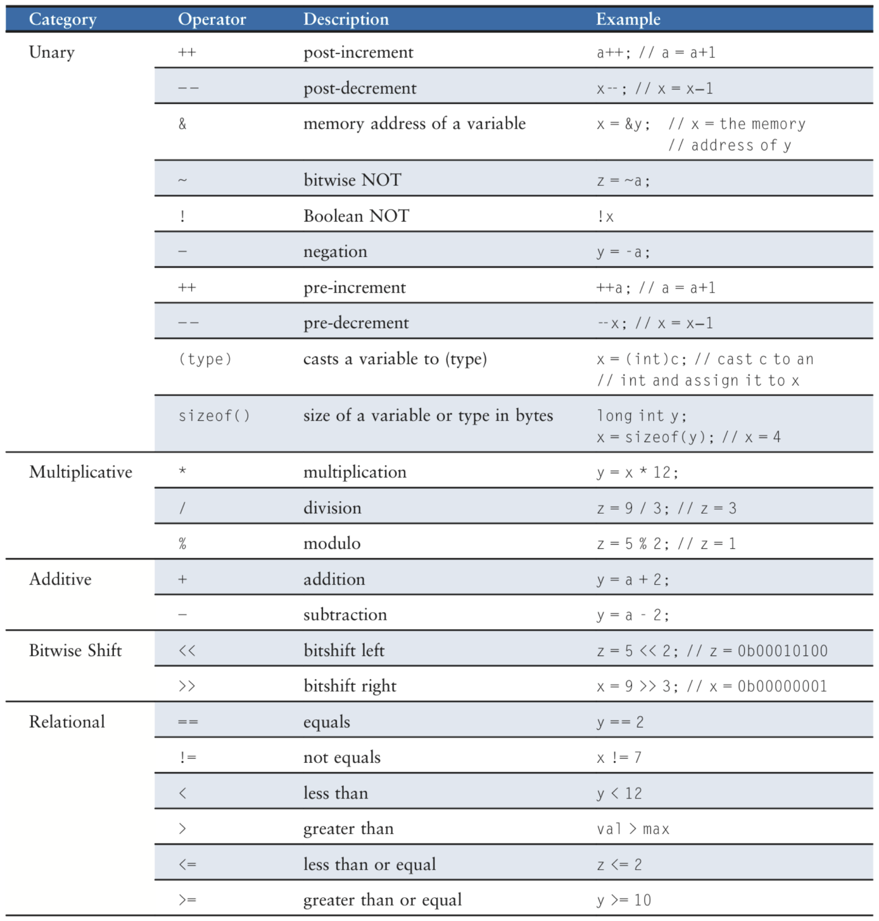


Table eC.2 **Primitive data types and sizes**

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`int16_t`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`uint16_t`	16	0	$2^{16} - 1 = 65{,}535$
`int32_t`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`uint32_t`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`int64_t`	64	-2^{63}	$2^{63} - 1$
`uint64_t`	64	0	$2^{64} - 1$
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Operators and Operator Precedence

Table eC.3 **Operators listed by decreasing precedence**



Category	Operator	Description	Example
Unary	`++`	post-increment	`a++; // a = a+1`
	`--`	post-decrement	`x--; // x = x-1`

	`&`	memory address of a variable	`x = &y;   // x = the memory`
			`// address of y`

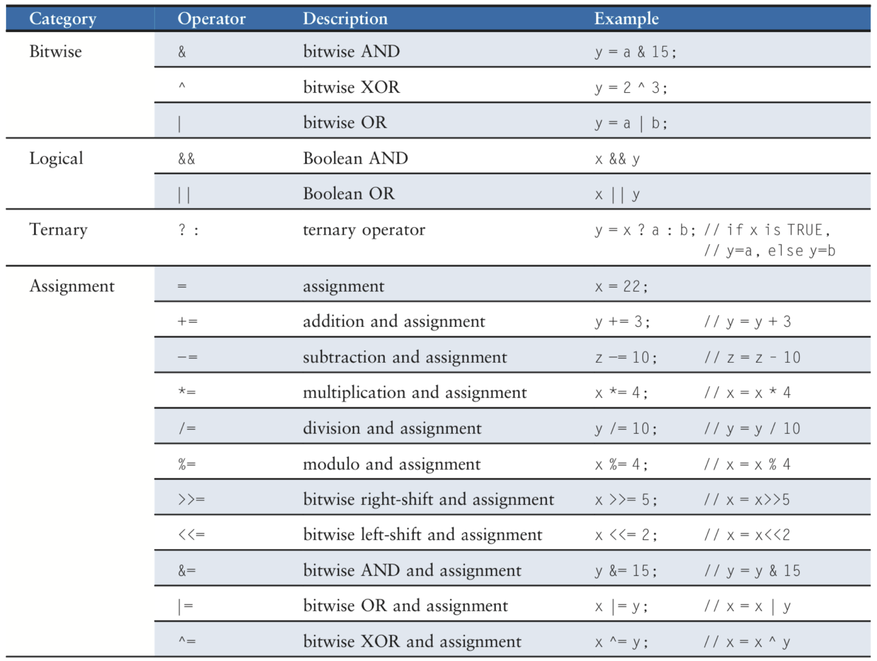
	`~`	bitwise NOT	`z = ~a;`
	`!`	Boolean NOT	`!x`
	`-`	negation	`y = -a;`
	`++`	pre-increment	`++a; // a = a+1`
	`--`	pre-decrement	`--x; // x = x-1`

	`(type)`	casts a variable to (type)	`x = (int)c; // cast c to an`
			`// int and assign it to x`

	`sizeof()`	size of a variable or type in bytes	`long int y;`
			`x = sizeof(y); // x = 4`

Multiplicative	`*`	multiplication	`y = x * 12;`
	`/`	division	`z = 9 / 3; // z = 3`
	`%`	modulo	`z = 5 % 2; // z = 1`
Additive	`+`	addition	`y = a + 2;`
	`-`	subtraction	`y = a - 2;`
Bitwise Shift	`<<`	bitshift left	`z = 5 << 2; // z = 0b00010100`
	`>>`	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	`==`	equals	`y == 2`
	`!=`	not equals	`x != 7`
	`<`	less than	`y < 12`
	`>`	greater than	`val > max`
	`<=`	less than or equal	`z <= 2`
	`>=`	greater than or equal	`y >= 10`

Table eC.3 **Operators listed by decreasing precedence—Cont'd**



Category	Operator	Description	Example
Bitwise	`&`	bitwise AND	`y = a & 15;`
	`^`	bitwise XOR	`y = 2 ^ 3;`
	`|`	bitwise OR	`y = a | b;`
Logical	`&&`	Boolean AND	`x && y`
	`||`	Boolean OR	`x || y`

Ternary	`? :`	ternary operator	`y = x ? a : b; // if x is TRUE,`
			`// y=a, else y=b`

Assignment	`=`	assignment	`x = 22;`
	`+=`	addition and assignment	`y += 3;     // y = y + 3`
	`-=`	subtraction and assignment	`z -= 10;    // z = z - 10`
	`*=`	multiplication and assignment	`x *= 4;     // x = x * 4`
	`/=`	division and assignment	`y /= 10;    // y = y / 10`
	`%=`	modulo and assignment	`x %= 4;     // x = x % 4`
	`>>=`	bitwise right-shift and assignment	`x >>= 5;    // x = x>>5`
	`<<=`	bitwise left-shift and assignment	`x <<= 2;    // x = x<<2`
	`&=`	bitwise AND and assignment	`y &= 15;    // y = y & 15`
	`|=`	bitwise OR and assignment	`x |= y;     // x = x | y`
	`^=`	bitwise XOR and assignment	`x ^= y;     // x = x ^ y`

Operator Precedence Tip!

You should only have to remember multiplication/division before addition/subtraction.
For everything else, use parentheses!

Important Keywords in C

- `volatile` – prevents the compiler from using a cached value (forces load)
- `const` – “read-only”. Prevents you from assigning a value to the variable.
- `static`
  - Inside a function: retains its values between calls.
  - Applied to a function: visible only in this file
- `extern`
  - Applied to a function definition: has global scope (redundant)
  - Applied to a variable: defined elsewhere
- `void`
  - As return type of function: doesn’t return a value
  - In a pointer declaration, the type of a generic pointer
  - In a parameter list: takes no parameters

Important Libraries

- `stdint.h` – standard fixed-width types (e.g., uint32_t)
- `stdio.h` - standard input and output. Contains functions like printf or fprintf.
- `stdlib.h` – standard library: random number generation (rand and srand), allocating or freeing memory (malloc and free).
- `math.h` – math library: standard math functions like sin, cos, sqrt, log, exp, floor, ceil.
- `string.h` – string library: functions to compare, copy, concatenate, and determine the length of a string.

Setting and Clearing Bits

C Idioms for Setting and Clearing Bits

```c
#define GPIOA_BASE 0x48000000
#define GPIOA_MODER (*((volatile unsigned long *) (GPIOA_BASE + 0x00)))

// Set bit 3 of the GPIOA_MODER to 1.

                                                                    ①
// Clear bit 7 of the GPIOA_MODER (i.e., set to 0)

                                                                    ②

```

① `GPIOA_MODER |= (1 << 3);`

② `GPIOA_MODER &= ~(1 << 7);`

Pointers and Arrays

Pointers and Arrays in C: Arrays

```
int * p = (int*) 0x20000000;

int a = *p;                                                   ①

int b = *(p+3);                                               ②

*(p+5) = b;                                                   ③
```

① Equivalent to `a= p[0]`

② Equivalent to `b= p[3]`, address `0x2000000C`

③ Equivalent to `p[5]= b`, address `0x20000014`

Pointers and Arrays in C: Strings

```c
char * str = (char *) 0x20001000;

str[13] = 'A'; ①
```

① Address `0x2000100D = 0x41`

Dereferencing

```c

int * p = (int*) 0x20000000;

int a = *p;                     ①
int * aptr = &a;                ②

*aptr = 3;                      ③

int * ptr = &p[0];              ④

ptr = &p[5]                     ⑤

*ptr = 42;                      ⑥
```

① Equivalent to a = `p[0]`

② `aptr` stores address of `a`

③ same as `a=3`

④ `ptr=0x200000000`

⑤ `ptr=0x20000014`

⑥ `p[5]=42`

Structures

```c
struct optional_tag {
    type_1 identifier_1;
    type_2 identifier_2;
    ...
    type_N identifier_N;
} optional_variable_definitions ;
```

Structures

```c
struct contact {
    char name[30];
    unsigned long long phone;
    float height;
};

struct contact jbrake; // example variable definition
```

How many bytes does this structure occupy in memory?

30*sizeof(char) + sizeof(unsigned long long) + sizeof(float)

= 30 B + 8 B + 4 B = 42 Bytes

Using structures as part of a new type

Can also wrap in a typedef to avoid needing to use the `struct` keyword.

```c
typedef struct my_tag {int i;} my_type; // Declaration of new type

// Creating variable with struct keyword


                                                                    ①

// Creating variable using new type

```

① `struct my_tag variable_1;`

② `my_type variable_2;`

Arrow Operator

Use a structure to access a chunk of memory in a specific location.

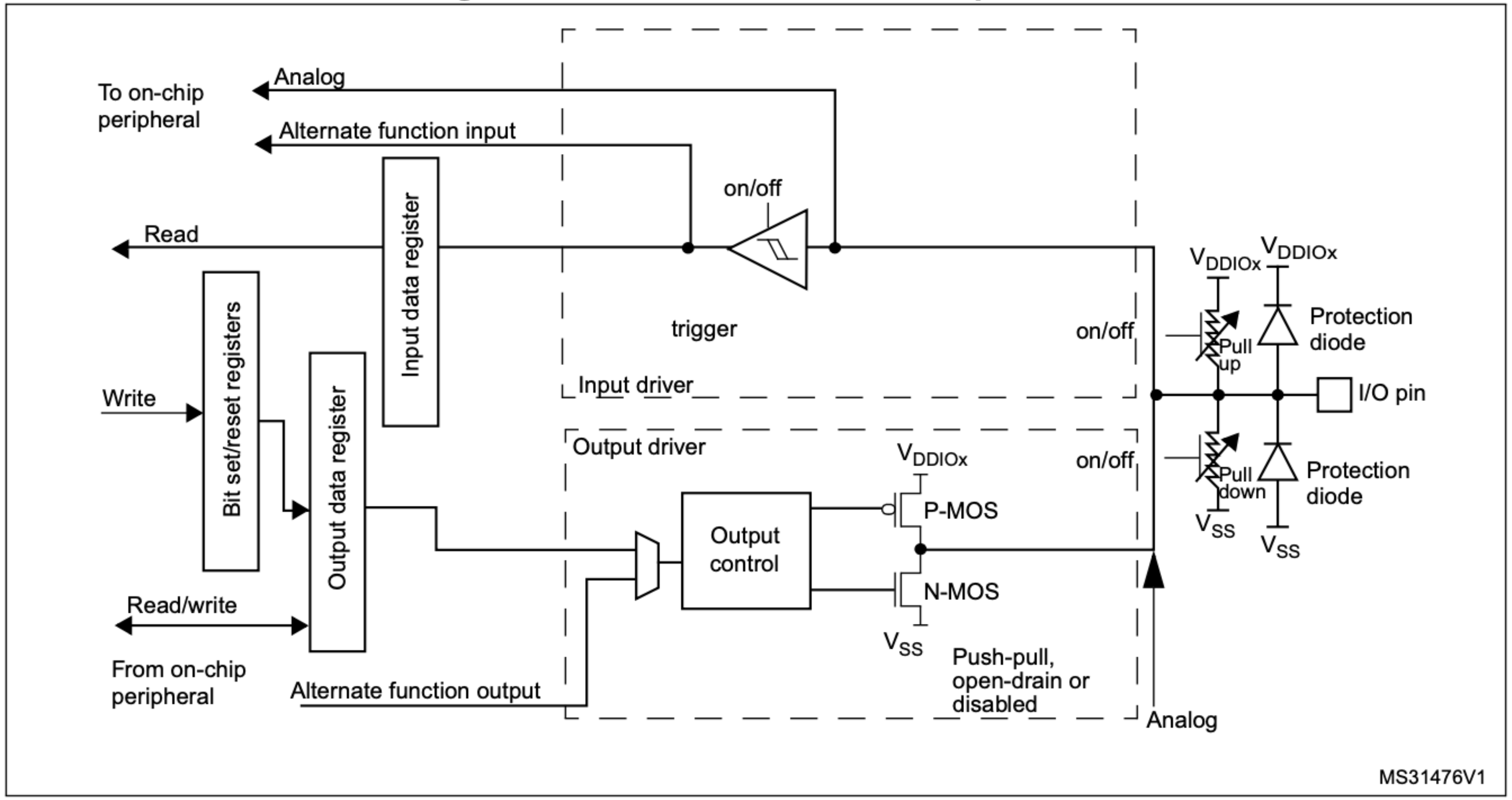
```c
typedef struct {
  char first_name[30];
  unsigned long long phone;
  float height;
} contact_type;

contact_type jbrake;
strcpy(jbrake.first_name, "Josh Brake");
jbrake.phone = (unsigned long long) 9096218553;
contact_type * contact_type_ptr = &jbrake;
unsigned long long phone_num = contact_type_ptr->phone;
```

Writing Device Drivers: GPIO Example

GPIO Block Diagram

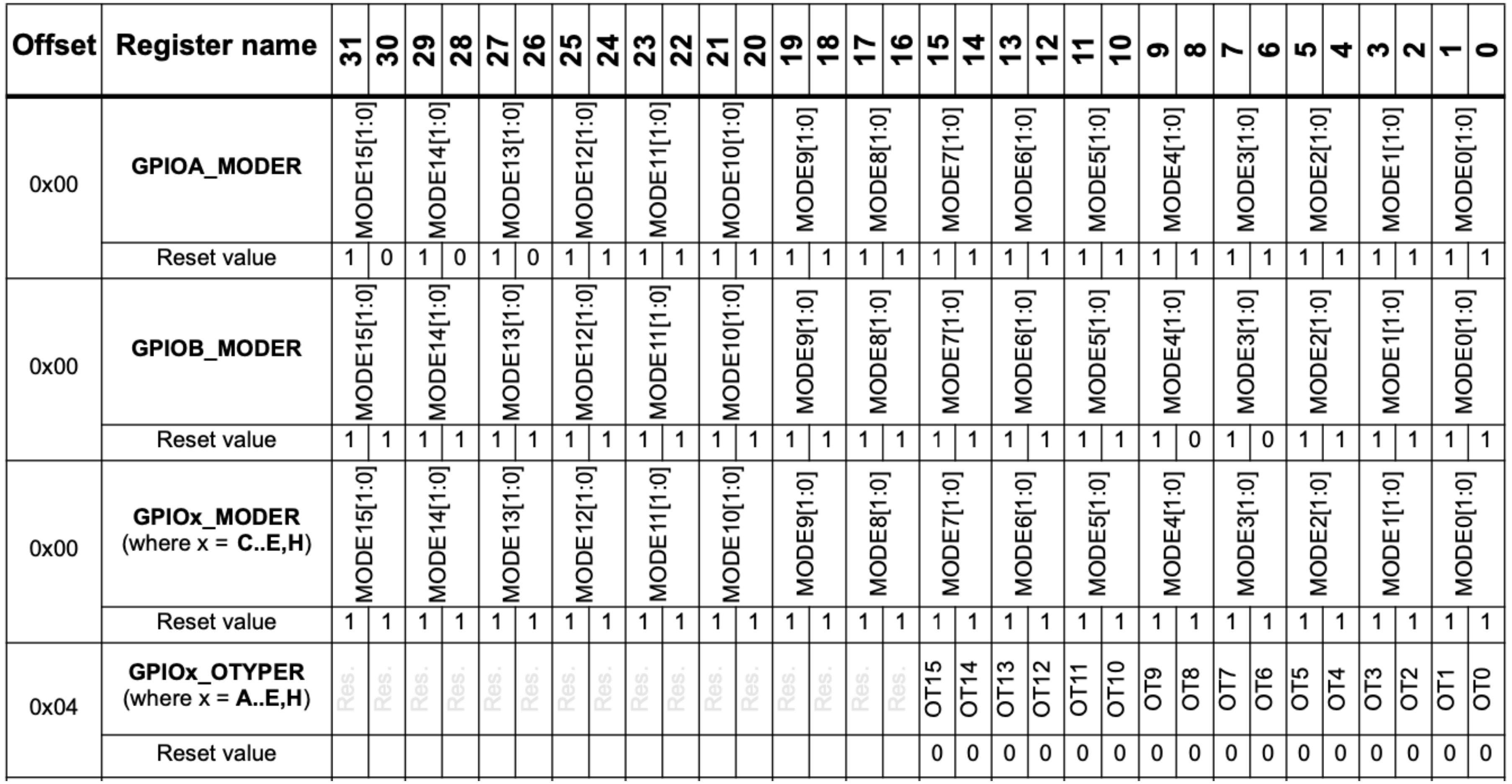
Figure 19. Basic structure of an I/O port bit



RM0394 p. 259

GPIO Register Map

Table 37. GPIO register map and reset values



Offset	Register name	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
0x00	**GPIOA_MODER**	MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]		MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
	Reset value	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x00	**GPIOB_MODER**	MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]		MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
	Reset value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1
0x00	**GPIOx_MODER** (where x = **C..E,H**)	MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]		MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
	Reset value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x04	**GPIOx_OTYPER** (where x = **A..E,H**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

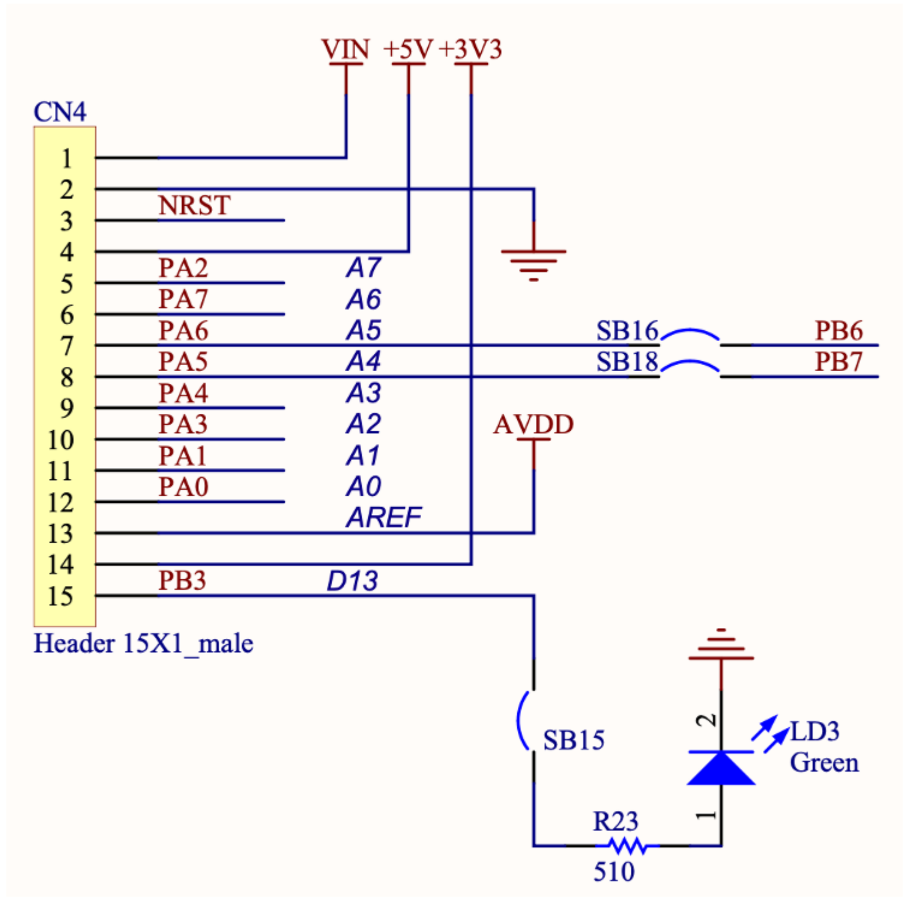
RM0394 p. 274

Steps for writing a device driver for a new peripheral

1. Look at block diagram
2. Note what elements in the diagram need to be configured
3. Find relevant registers and bits
4. Write code
    1. Base address for peripheral
    2. Create structure to define registers

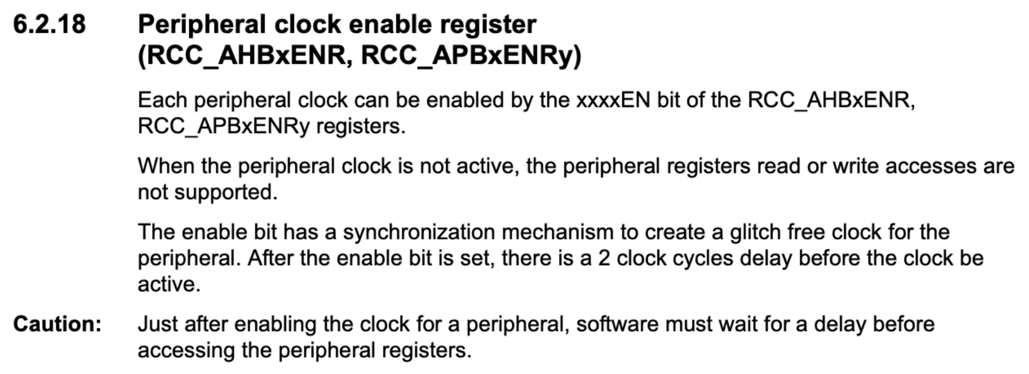
Blink LED

On-board LED connected to pin PB3.



UM1956 p. 33

Enabling peripheral clock



6.2.18 Peripheral clock enable register (RCC_AHBxENR, RCC_APBxENRy)

Each peripheral clock can be enabled by the xxxxEN bit of the RCC_AHBxENR, RCC_APBxENRy registers.

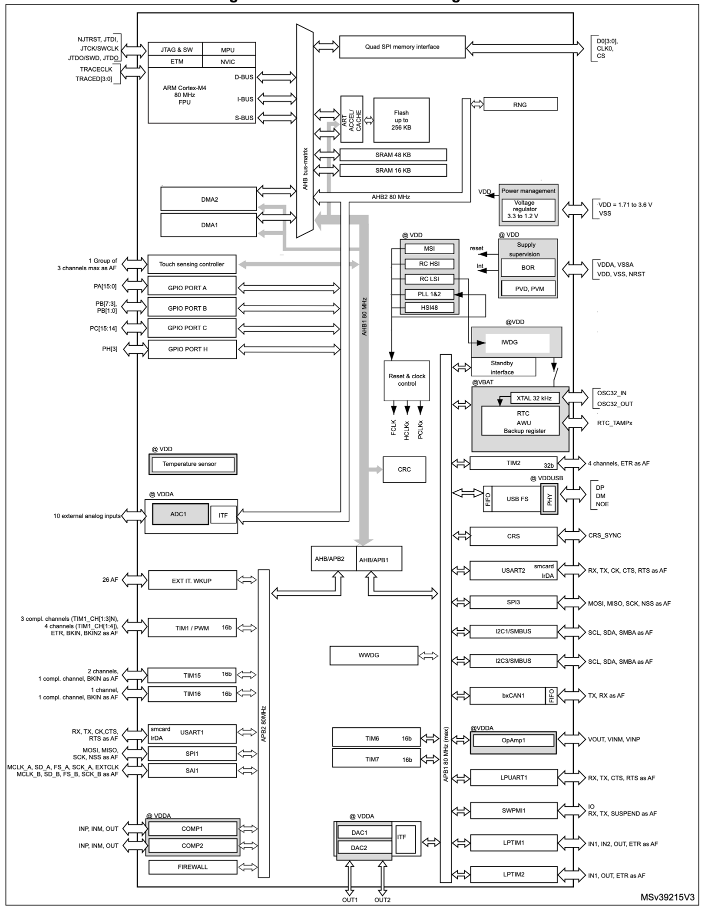
When the peripheral clock is not active, the peripheral registers read or write accesses are not supported.

The enable bit has a synchronization mechanism to create a glitch free clock for the peripheral. After the enable bit is set, there is a 2 clock cycles delay before the clock be active.

Caution: Just after enabling the clock for a peripheral, software must wait for a delay before accessing the peripheral registers.

RM0394 p. 191

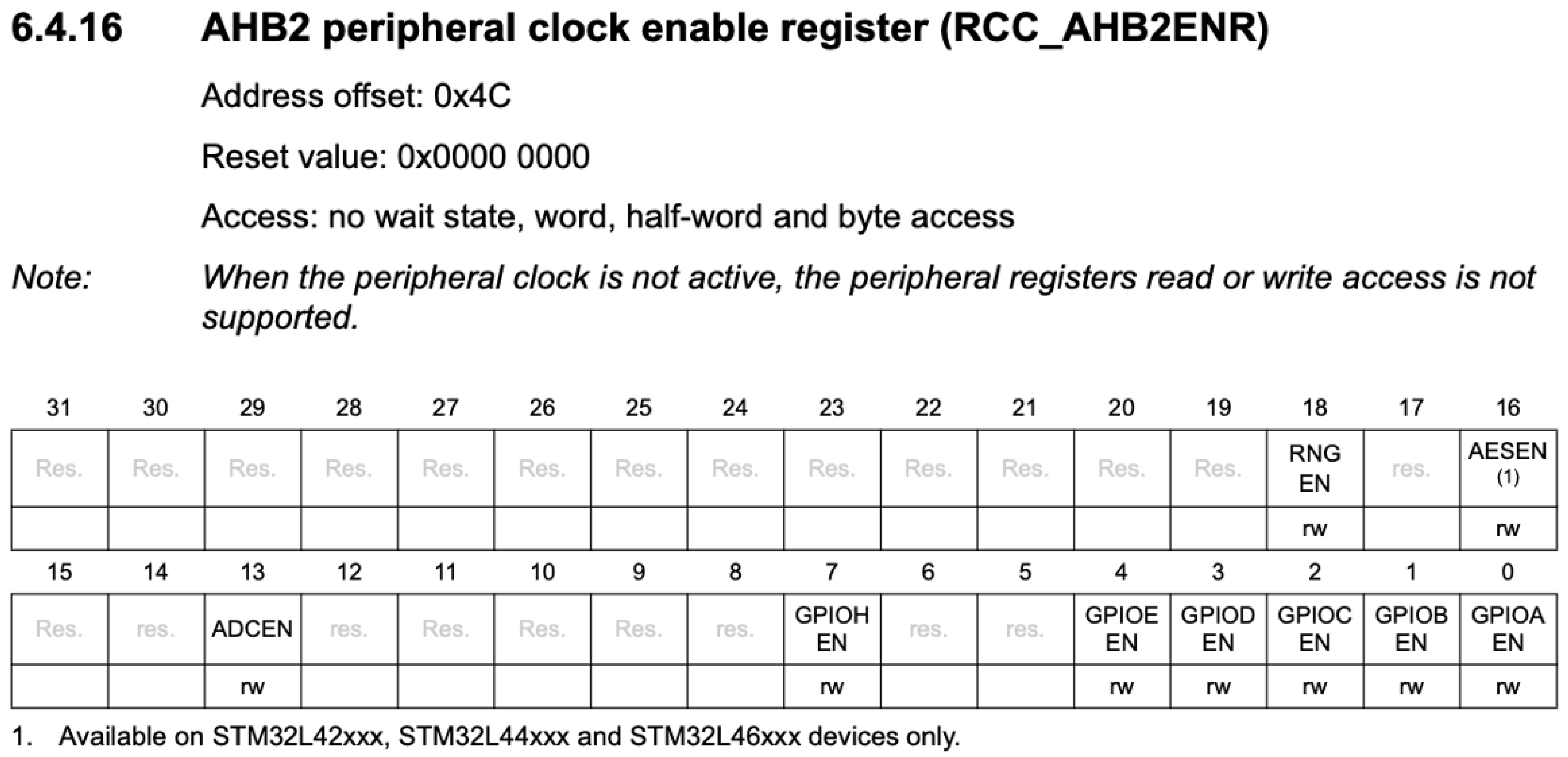
Figure 1. STM32L432xx block diagram



Note: AF: alternate function on I/O pins.

DS11451 p. 13

Finding clock enable bit for GPIOB



6.4.16 AHB2 peripheral clock enable register (RCC_AHB2ENR)

Address offset: 0x4C

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access

Note: When the peripheral clock is not active, the peripheral registers read or write access is not supported.

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.	RNG EN	res.	AESEN [1]
													rw		rw

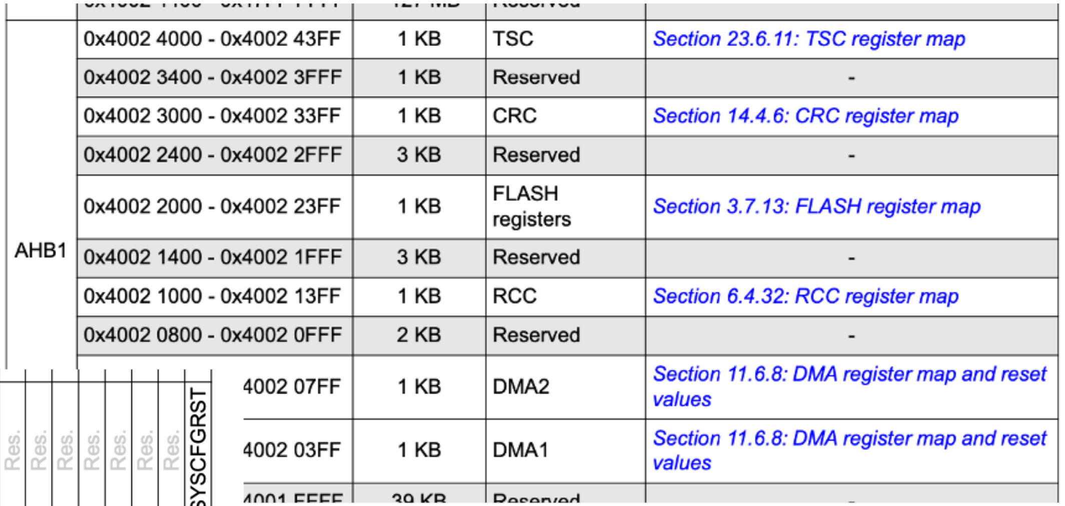
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	res.	ADCEN	res.	Res.	Res.	Res.	res.	GPIOH EN	res.	res.	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
		rw						rw			rw	rw	rw	rw	rw

1. Available on STM32L42xxx, STM32L44xxx and STM32L46xxx devices only.

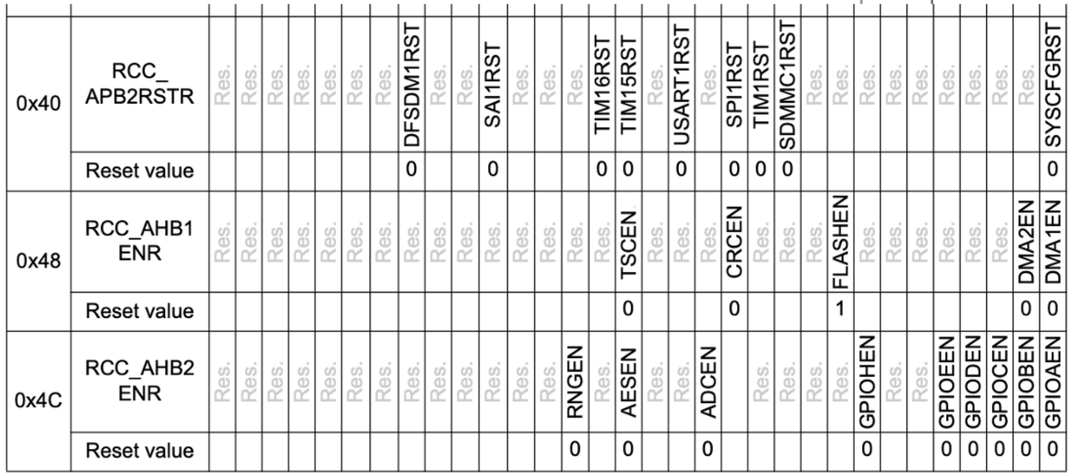
RM0394 p. 218

Where is that bit located?

1. Look in system and memory overview section
2. Look RCC register mapping to find register and bit offsets



AHB1	0x4002 4000 - 0x4002 43FF	1 KB	TSC	*Section 23.6.11: TSC register map*
	0x4002 3400 - 0x4002 3FFF	1 KB	Reserved	-
	0x4002 3000 - 0x4002 33FF	1 KB	CRC	*Section 14.4.6: CRC register map*
	0x4002 2400 - 0x4002 2FFF	3 KB	Reserved	-
	0x4002 2000 - 0x4002 23FF	1 KB	FLASH registers	*Section 3.7.13: FLASH register map*
	0x4002 1400 - 0x4002 1FFF	3 KB	Reserved	-
	0x4002 1000 - 0x4002 13FF	1 KB	RCC	*Section 6.4.32: RCC register map*
	0x4002 0800 - 0x4002 0FFF	2 KB	Reserved	-
	4002 07FF	1 KB	DMA2	*Section 11.6.8: DMA register map and reset values*
	4002 03FF	1 KB	DMA1	*Section 11.6.8: DMA register map and reset values*
	4001 FFFF	39 KB	Reserved	-



0x40	RCC_ APB2RSTR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DFSDM1RST	Res.	Res.	SAI1RST	Res.	Res.	Res.	TIM16RST	TIM15RST	Res.	USART1RST	Res.	SPI1RST	TIM1RST	SDMMC1RST	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SYSCFGRST
	Reset value								0			0				0	0		0		0	0	0										0
0x48	RCC_AHB1 ENR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	Res.	Res.	CRCEN	Res.	Res.	Res.	FLASHEN	Res.	Res.	Res.	Res.	Res.	Res.	DMA2EN	DMA1EN
	Reset value																0				0				1							0	0
0x4C	RCC_AHB2 ENR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNGEN	Res.	AESEN	Res.	Res.	ADCEN		Res.	Res.	Res.	Res.	GPIOHEN	Res.	Res.	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
	Reset value														0		0			0						0			0	0	0	0	0

Configuration steps to enable basic GPIO: RCC

Turn on clock domain in Reset and Clock Control (RCC)

`RCC` base address: `0x40020400`

`RCC_AHB2ENR` register offset: `0x4C`

Bit for `GPIOB_EN`: `1`

Configure pin as output in GPIO register block

Configure pin as an output (GPIO_MODER)

Base address of `GPIOB`: `0x48000400`

Offset of MODER register: `0x00`

Bits in MODER to be set: 6 and 7

Value for relevant bits to configure pin as output: `01`

Blink Demo: Includes

```c
// Nucleo-L432KC Blink demo
// Josh Brake
// jbrake@hmc.edu
// 9/21/22
#include <stdint.h>
#define GPIOB_BASE_ADR (0x48000400UL)
#define RCC_BASE_ADR (0x40021000UL)
#define RCC_AHB2ENR ((uint32_t *) (RCC_BASE_ADR + 0x4C))
#define GPIOB_MODER ((uint32_t *) (GPIOB_BASE_ADR + 0x00))
#define GPIOB_ODR ((uint32_t *) (GPIOB_BASE_ADR + 0x14))
#define DUMMY_DELAY 100000

...
```

Blink Demo: `main`

```c
...

int main(void) {
  // Initialization code
  *RCC_AHB2ENR |= (1 << 1);
  // Set PB3 as output (MODER bit 7 to 0 and bit 6 to 1)
  *GPIOB_MODER &= ~(1 << 7);
  *GPIOB_MODER |= (1 << 6);
while(1) {
  for(volatile int i = 0; i < DUMMY_DELAY; i++);
  *GPIOB_ODR ^= (1 << 3);
}
```

Miscellanous Notes

Using MCU while connected to development board

- Make sure that you have the `MCU_+5V` header connected. This ensures the on-board voltage regulators work which makes sure the reset signal is held high. If not, you won't be able to connect to your MCU to program it (the reset pin will float and the MCU will always be in reset!)
- Remove jumper that came installed by default on the Nucleo board (connects reset to ground!)

Using structures to model memory-mapped I/O

```c
// Base addresses for GPIO ports
#define GPIOA_BASE (0x48000000U)
typedef struct
{
__IO uint32_t MODER;    /*!< GPIO port mode register, Address offset: 0x00 */
__IO uint32_t OTYPER;   /*!< GPIO port output type register, Address offset: 0x04 */
__IO uint32_t OSPEEDR;  /*!< GPIO port output speed register, Address offset: 0x08 */
__IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register, Address offset: 0x0C */
__IO uint32_t IDR;  /*!< GPIO port input data register, Address offset: 0x10 */
__IO uint32_t ODR;  /*!< GPIO port output data register, Address offset: 0x14 */
__IO uint32_t BSRR;     /*!< GPIO port bit set/reset register, Address offset: 0x18 */
__IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C */
__IO uint32_t AFR[2];   /*!< GPIO alternate function registers, Address offset: 0x20-0x24 */
__IO uint32_t BRR;  /*!< GPIO Bit Reset register, Address offset: 0x28 */
} GPIO_TypeDef;
```

`__IO` is defined with a `#define` statement to one of the C keywords we discussed earlier. Which one?

Wrap Up

- C is libertarian – will allow you to do many things, not all of which are good for you.
- Understanding certain C data structures like pointers and structures will enable you to more easily and naturally write code to control your MCU.
- The MCU reference manual contains the information needed to write code to configure and manipulate the peripherals using memory-mapped I/O.