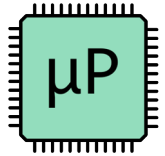
C Programming

Lecture 2

Microprocessor-based Systems (E155)

Prof. Josh Brake

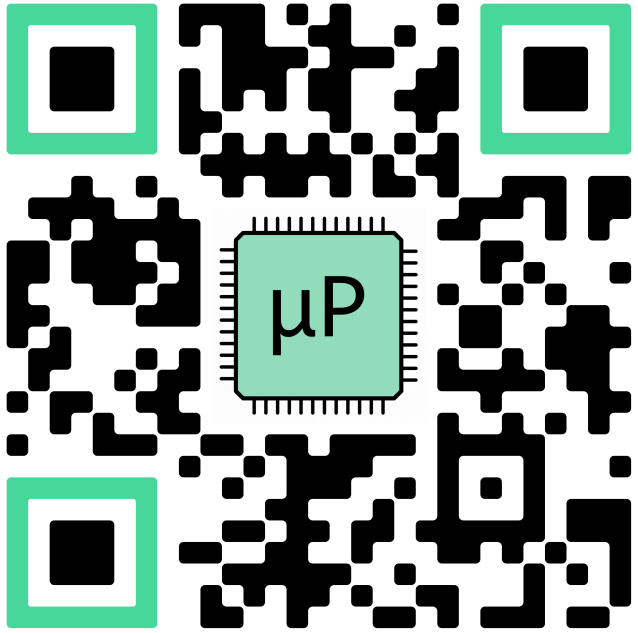


Outline

- Background and history
- Review of C concepts
  - Data types
  - Comments
  - Operators
  - Global and local variables
  - Control flow constructs
  - Pointers
  - Arrays
  - Characters
  - Strings
  - Inline assembly
  - Direct hardware access
  - Libraries
- Building a program
- Lab 1 Hints

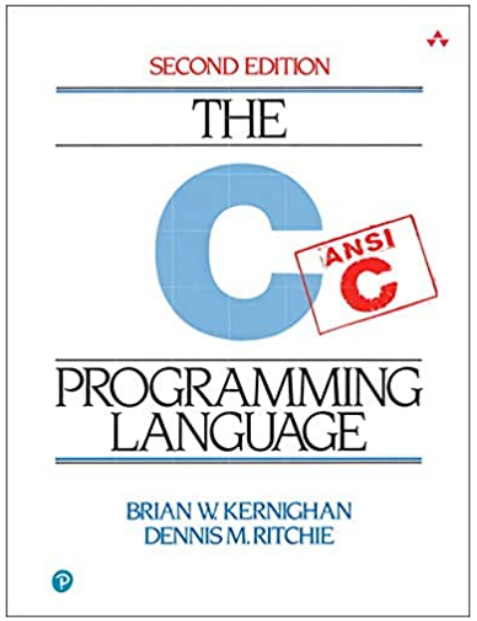
C Review Quiz!

https://pollev.com/joshbrake155



Background

- Formally introduced in 1978 in *The C Programming Language* by Kernighan and Ritchie.
- In 1989 the American National Standards Institute (ANSI) expanded and standardized the language which became known as ANSI C, Standard C, or C89.
- The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) adopted the standard shortly after and updated the standard in 1999 to C99.
- Since C99, there have been two new standards introduced: C11 and C18. C2X is currently in development. Here



Why C?

- Code efficiency
- Low-level control
- More readable and portable than assembly
- Many free C compilers available
- Can be very close to the same performance as optimized assembly

Review DDCA ARMed Chapter 9

Data types on a 32-bit architecture

Type	Size in bits	Natural alignment in bytes	Range of values
char	8	1 (byte-aligned)	0 to 255 (unsigned) by default. –128 to 127 (signed) when compiled with --signed_chars.
signed char	8	1 (byte-aligned)	–128 to 127
unsigned char	8	1 (byte-aligned)	0 to 255
(signed) short	16	2 (halfword-aligned)	–32,768 to 32,767
unsigned short	16	2 (halfword-aligned)	0 to 65,535
(signed) int	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
unsigned int	32	4 (word-aligned)	0 to 4,294,967,295
(signed) long	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
unsigned long	32	4 (word-aligned)	0 to 4,294,967,295
(signed) long long	64	8 (doubleword-aligned)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	64	8 (doubleword-aligned)	0 to 18,446,744,073,709,551,615
float	32	4 (word-aligned)	1.175494351e-38 to 3.40282347e+38 (normalized values)
double	64	8 (doubleword-aligned)	2.2250738585072014e-308 to 1.7976931348623157e+308 (normalized values)
long double	64	8 (doubleword-aligned)	2.2250738585072014e-308 to 1.7976931348623157e+308 (normalized values)
All pointers	32	4 (word-aligned)	Not applicable.

Data types on 32-bit architecture

Be careful, sizes are implementation specific! The C standard only guarantees that a byte is *at least* 8 bits.

Solutions:

- Use `<stdint.h>` for defined integer sizes. (e.g., defines `int`*N*`_t` for an *N* bit signed, twos-complement integer)
- Check data sizes with `sizeof()`
- Use `CHAR_BIT` macro in `<climits.h>`

Comments

Comments can begin with /* and end with */. They can span multiple lines. Comments can also begin with // and terminate at the end of the line.

```
// this is an example of a one-line comment.

/* this is an example
   Of a multi-line comment */
```

Operators

Decreasing Precedence

e.g., Multiplication is evaluated before addition, which is evaluated before bitwise shifts.

When in doubt, use parentheses to specify what you mean!

Category	Name	Symbol
Monadic	Post-increment	++
	Post-decrement	--
	Address	&
	Bitwise NOT	~
	Type cast	(type)
	Logical NOT	!
	Negation	-
	Pre-increment	++
	Pre-decrement	--
	Size of data	sizeof()
Multiplicative	Modulus	%
	Multiply	*
	Divide	/
Additive	Add	+
	Subtract	-
Bitwise Shift	Shift left	<<
	Shift right	>>
Relational	Less than	<
	Less than or equal	<=
	Greater than	>
	Greater than or equal	>
	Equal to	==
	Not equal to	!=
Bitwise	AND	&
	XOR	^
	OR	|
Logical	AND	&&
	OR	||
Ternary	Conditional	? :
Assignment	Equal	=
	Other arithmetic	+=, -=, *=, /=, %=
	Other shift	>>=, <<=
	Other bitwise	&=, |=, ^=

Global and local variables

- A global variable is declared outside of all the functions (normally at the top of a program) and can be accessed by all functions.
- A local variable is declared inside a function and can only be used by that function.
- Use global variables sparingly because they make large programs more difficult to read.

Local Variables

```
int test_func(void) {
  int a = 5;
  int b = 7;

  int c = a + b;
  return c;
};
```

Global Variables

```
int a = 5;
int b = 7;

int test_func(void) {
  int c = a + b;
  return c;
};
```

Control flow structures

- if/else
- while loops
- do-while loops
- for loops

if/else

```
int bigger(int a, int b) {
       if (a > b) return a;
       else return b;
}
```

While loops

```
void main(void) {
	int i = 0, sum = 0;

	// add the numbers from 0 to 9
	while (i < 10) { // while loops check condition before executing body
		sum = sum + i;
		i++;
	}
}
```

Do-while loops

```
void main(void) {
	int i = 0, sum = 0;

	// add the numbers from 0 to 9
	do {
		sum = sum + i;
		i++;
	} while (i < 10); // do loops check condition after executing body
}
```

For loops

```
void main(void) {
	int i;
	int sum = 0;

	// add the numbers from 0 to 9
	for (i=0; i<10; i++) {
		sum += i;
	}
}
```

Pointers

- A pointer is the address of a variable.
- How big is a pointer?
- In a variable declaration, a * before a variable name indicates that the variable is a pointer to the declared type.
- In a variable use, the * operator dereferences a pointer, returning the value at the given address. The & operator is read “address of,” giving the address of the variable being referenced.

Pointer example

```
unsigned long salary1, salary2;  // 32-bit numbers
unsigned long *ptr;              /* a 32-bit pointer specifying the address of an
                                 unsigned long variable */
```

The compiler will assign arbitrary locations in RAM for these variables. For the sake of concreteness, suppose `salary1` is at addresses `0x40-43`, `salary2` is at addresses `0x44-47` and `ptr` is at `0x48-51`.

```
salary1 = 67500;           // assign salary1 to be $67500 = 0x000107AC
ptr = &salary1;            // assign ptr to be 0x0040, the address of salary1
salary2 = *ptr + 1000;     /* dereference ptr to give the contents of address 40 =
                              67500, then add $1000 and set salary2 to $68500 */
```

Pointer example

RAM Address	Contents	Notes
0x040	AC	LSB of salary1
0x041	07	
0x042	01	
0x043	00	MSB of salary1
0x044	94	LSB of salary2
0x045	0B	
0x046	01	
0x047	00	MSB of salary2
0x048	40	LSB of ptr
0x049	00	
0x050	00	
0x051	00	MSB of ptr

Little endian

Arrays

- An array is a group of variable stored in consecutive addresses in memory.
- An array is referred to by the address of the 0^{th} element.
- Warning! The programmer is responsible for making sure they don't access elements beyond the limits of the array. The code will compile fine but give you weird bugs!

Array Example

In the following example, suppose we have an array indicating how many wombats crossed the road each hour for each of the past 10 hours. Suppose the 0^{th} element is stored at address 0x20.

```
int wombats[10];              // array of 10 4-byte quantities stored at 0x20-0x47.
int *wombptr;                 // a pointer to an integer

wombats[0] = 342;             // store 342 in addresses 0x20-23
wombats[1] = 9;               // store 9 in addresses 0x24-27
wombptr = &wombats[0];        // wombptr = 0x020
*(wombptr+4) = 7;             /* offset of 4 elements, or 16 bytes. Thus addresses 0x36-39 = 7,
                              so this is another way to write wombats[4] = 7. */
```

Characters

- Characters (`char`) are byte sized variables. On almost every system they will be 8-bits.
- Can be viewed either as a number between -128 and 127 or as an ASCII code for a letter, digit, symbol, etc.
- Can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character in single quotes.
- For example, the letter A has the ASCII code `0x41`, B=`0x42`, etc. Thus `'A' + 3` is `0x44`, or `'D'`.
- Special characters include:
  - `'\r'`: carriage return (when you press the enter key)
  - `'\n'`: new line
  - `'\t'`: tab
  - `0x00` : null character used to terminate strings

Strings

- A string is an array of characters.
- Each character is a byte representing the ASCII code.
- The array size is the maximum size, but the actual length could be shorter.
- In C, the length of the string is determined by looking for a null character (`0x00`) at the end of the string.

```
void strcpy(char *src, char *dst)
{
	int i = 0;

	do {
		dst[i] = src[i];		// copy characters one byte at a time
	} while (src[i++] != NULL);	// until the NULL terminator is found
}
```

Inline assembly

- Occasionally you might want to include assembly code in parts of your program.
- It is possible with the GNU toolchain to include inline assembly, but you should use it sparingly as the compiler will not optimize your code.
- Begin a block of assembly code with *__asm__ or asm.*

General Format

```
__asm__(
code
: output operand list
: input operand list
: clobber list
);
```

NOP example

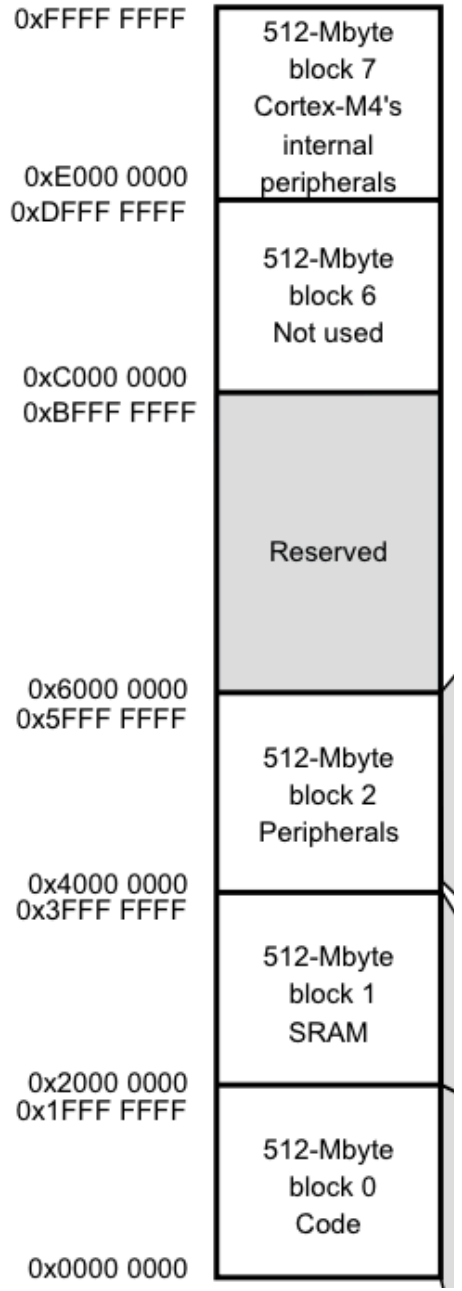
```
__asm__("mov r0,r0");
```

http://www.ethernut.de/en/documents/arm-inline-asm.html

Direct hardware access

- You may access most special function registers by the names given in the MCU's user manual by including the header file `stm32f401xe.h`. (We'll talk more about this when we discuss the Cortex Microcontroller Software Interface Standard)

```
#define FLASH_BASE 0x08000000UL /*!< FLASH(up to 1 MB) base address in the alias region */
#define SRAM1_BASE 0x20000000UL /*!< SRAM1(96 KB) base address in the alias region */
#define PERIPH_BASE 0x40000000UL /*!< Peripheral base address in the alias region */
#define SRAM1_BB_BASE 0x22000000UL /*!< SRAM1(96 KB) base address in the bit-band region */
#define PERIPH_BB_BASE 0x42000000UL /*!< Peripheral base address in the bit-band region */
#define BKPSRAM_BB_BASE 0x42480000UL /*!< Backup SRAM(4 KB) base address in the bit-band region */
#define FLASH_END 0x0807FFFFUL /*!< FLASH end address */
#define FLASH_OTP_BASE 0x1FFF7800UL /*!< Base address of : (up to 528 Bytes) embedded FLASH OTP Area */
#define FLASH_OTP_END 0x1FFF7A0FUL /*!< End address of : (up to 528 Bytes) embedded FLASH OTP Area */
```



STM32F401RE Datasheet p. 51, Fig. 15

Libraries

- Libraries in C are combination of header (`.h`) and source (`.c`) or partially compiled object (`.o`) files which provide various functions.
- Headers provide function declarations and macro definitions.
- Source code provides function implementations.
- Some frequently used C libraries
  - `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.

Q: Why do we need two separate files for the code?

Structures

- Structures are used to store a collection of data of various types

Basic Declaration

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

Example

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};
```

GPIO Structure from STM32 Registers

Table 27. GPIO register map and reset values

Offset	Register	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	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	GPIOx_ OTYPER (where x = A..E and H)	Reserved																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
0x08	GPIOA_ OSPEEDER	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	Reset value	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

STM32F401RE Reference Manual p.164

```
#define __IO volatile

    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 */
    } GPIO_TypeDef;
```

Common C idioms for low-level access

```
#define GPIOA_BASE 0x40020000
#define GPIOA_MODER (*((volatile unsigned long *) (GPIOA_BASE + 0x00)))
```

- Setting/clearing bits

`GPIOA_MODER |= (1 << 3)`	Set bit 3 of the GPIOA_MODER to 1.
`GPIOA_MODER &= ~(1 << 7)`	Clear bit 7 of the GPIOA_MODER (i.e., set to 0)

See Yiu Chapter 2.7

Common C idioms for low-level access

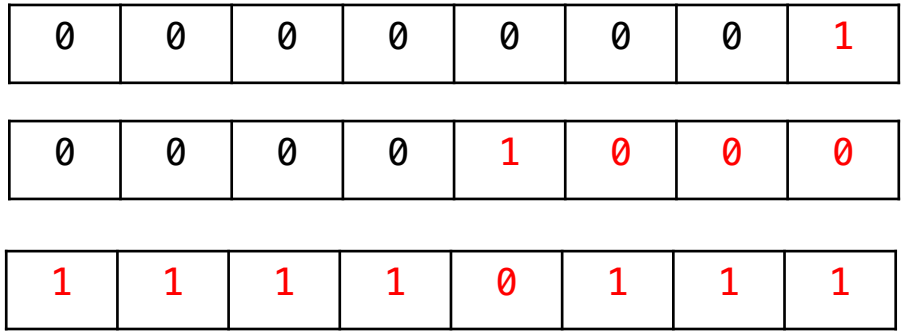
```
#define GPIOA_BASE 0x40020000
#define GPIOA_MODER (*((volatile unsigned long *) (GPIOA_BASE + 0x00)))
```

- Setting/clearing bits

`GPIOA_MODER |= (1 << 3)` Set bit 3 of the GPIOA_MODER to 1.

`GPIOA_MODER &= ~(1 << 7)` Clear bit 7 of the GPIOA_MODER (i.e., set to 0)

A few examples…



|= vs. &=

Accessing values from structs

`<ptr>-><member>` is equivalent to `(*<ptr>).<member>`

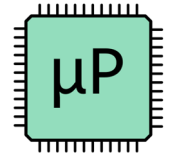
```
// Base addresses for GPIO ports
#define GPIOA_BASE (0x40020000U)

// GPIO register structs here
typedef struct {
volatile uint32_t GPIO_MODER; // GPIO Offset 0x00 GPIO port mode register
volatile uint32_t GPIO_OTYPER; // GPIO Offset 0x04
volatile uint32_t GPIO_OSPEEDR; // GPIO Offset 0x08
volatile uint32_t GPIO_PURPDR; // GPIO Offset 0x0C
volatile uint32_t GPIO_IDR; // GPIO Offset 0x10
volatile uint32_t GPIO_ODR; // GPIO Offset 0x14
volatile uint32_t GPIO_BSRR; // GPIO Offset 0x18
volatile uint32_t GPIO_LCKR; // GPIO Offset 0x1C
volatile uint32_t GPIO_AFRL; // GPIO Offset 0x20
volatile uint32_t GPIO_AFRH; // GPIO Offset 0x24
} GPIO;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define GPIOA ((GPIO*) GPIOA_BASE)

// Read value
int pin_value = ((GPIOA->GPIO_IDR) >> pin) & 1;
```

Lab 1 Hints/Questions



Learning Objectives

By the end of this lab you will have:

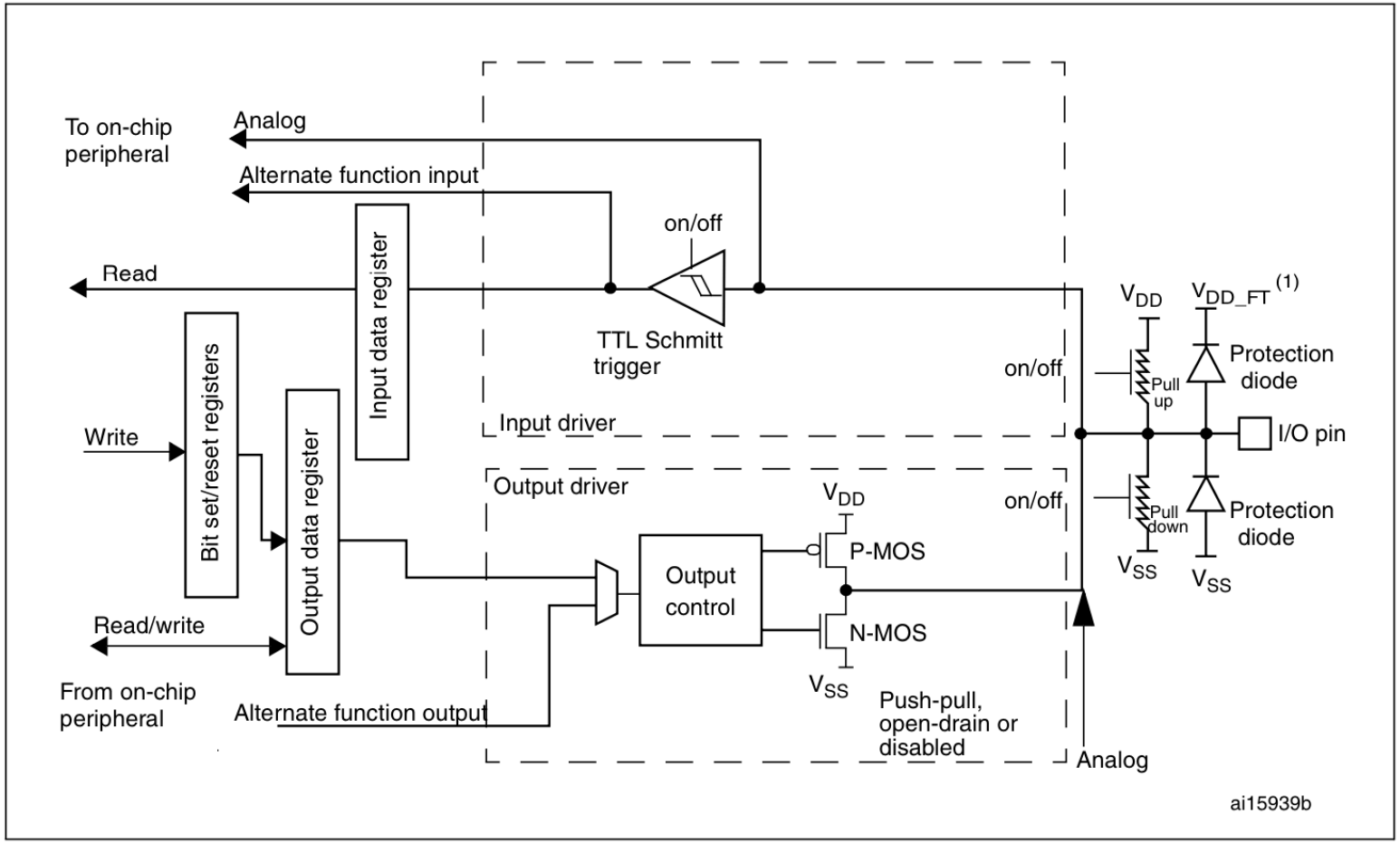
- Configured a Docker container with a working GNU toolchain to target ARM devices
- Compiled and uploaded bare metal C and ARM assembly code on the STM32F401RE development board
- Written a makefile to simplify the process of compiling programs
- Debugged a program using the GNU debugger GDB
- Written a simple C library to control GPIO pins and blink an onboard LED on the Nucleo board

Requirements

- Write a library in C to control the GPIO pins on your board. Your library should contain the following functions and use bitfield structs to interact with the memory-mapped registers for the GPIO pins:
  - pinMode
  - digitalWrite
  - digitalRead
- Write a main function to blink the onboard user LED LD2 at ~2 Hz. (Hint: search in the documentation to find what GPIO pin this LED is connected to.)
- Write a makefile to automate the building of your source code.
- Use the oscilloscope functionality of your ADALM2000 to measure the exact LED blinking frequency.

Lab 1 GPIO structure

Figure 16. Basic structure of a five-volt tolerant I/O port bit



STM32F401RE Reference Manual p. 147

Lab 1 Hints

- Refer to the documentation to see what registers need to be configured (reference manual is your friend!).
- Steps to initialize GPIO as output
  1. Turn on clock to peripheral by configuring the peripheral clock settings in the Reset and clock control (RCC) settings. To do this, set the right bit to 1 in RCC_AHB1ENR.
  2. Configure pin mode using GPIOA_MODER register to set the pin as an output (`0b01`).
  3. Write the output using either the output data register (GPIOx_ODR) or the port bit set/reset register (GPIOx_BSRR).

GPIO Mode Register

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..E and H)

Address offset: 0x00

Reset values:

- 0x0C00 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

Interfacing with the board

- Find GPIO pin that LD1 (user LED) is connected to (Nucleo user manual p. 23)

6.4 LEDs

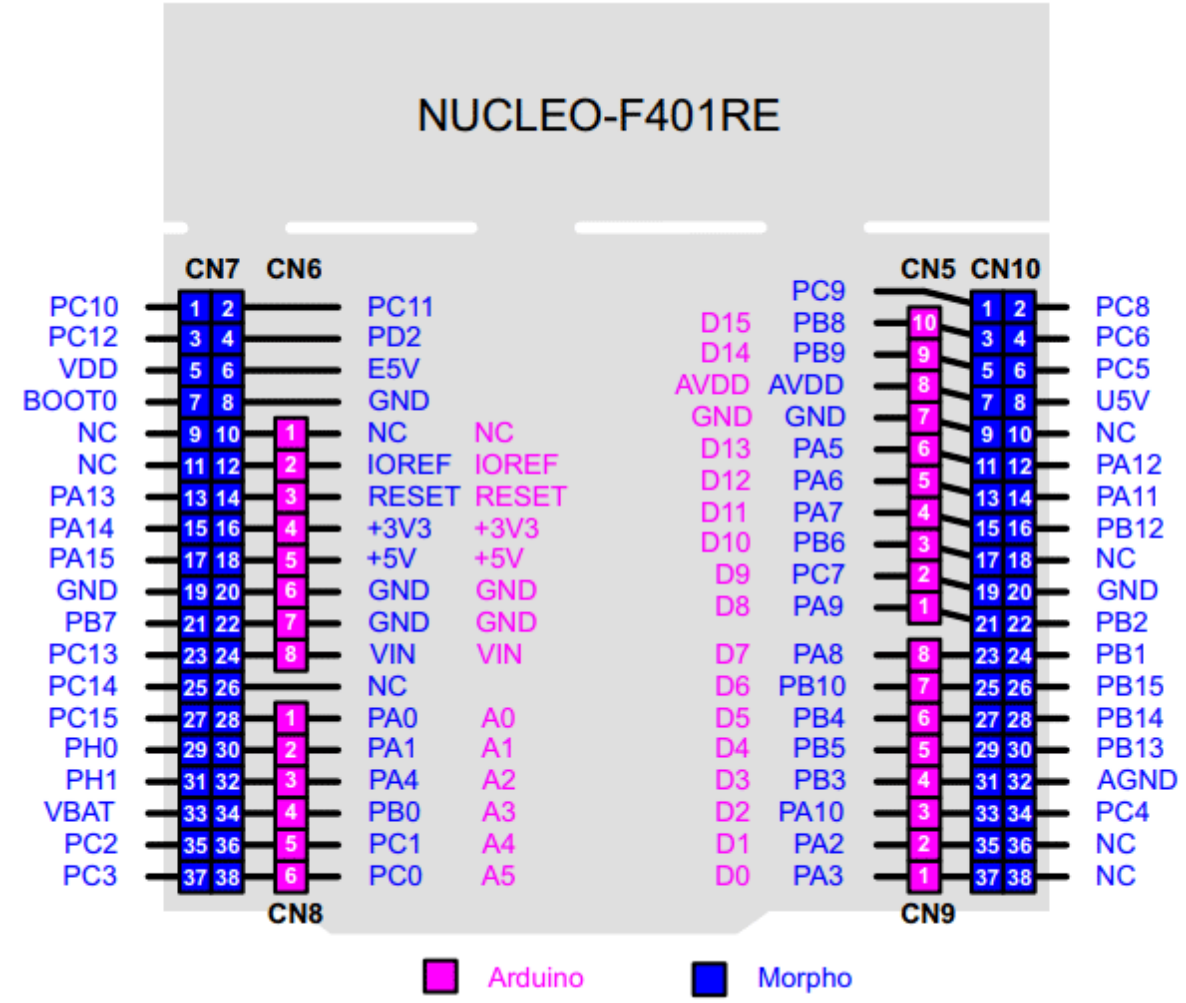
The tricolor LED (green, orange, red) LD1 (COM) provides information about ST-LINK communication status. LD1 default color is red. LD1 turns to green to indicate that communication is in progress between the PC and the ST-LINK/V2-1, with the following setup:

- Slow blinking Red/Off: at power-on before USB initialization
- Fast blinking Red/Off: after the first correct communication between the PC and ST-LINK/V2-1 (enumeration)
- Red LED On: when the initialization between the PC and ST-LINK/V2-1 is complete
- Green LED On: after a successful target communication initialization
- Blinking Red/Green: during communication with target
- Green On: communication finished and successful
- Orange On: Communication failure

User LD2: the green LED is a user LED connected to Arduino signal D13 corresponding to STM32 I/O PA5 (pin 21) or PB13 (pin 34) depending on the STM32 target. Refer to *Table 11* to *Table 23* when:

- the I/O is HIGH value, the LED is on
- the I/O is LOW, the LED is off

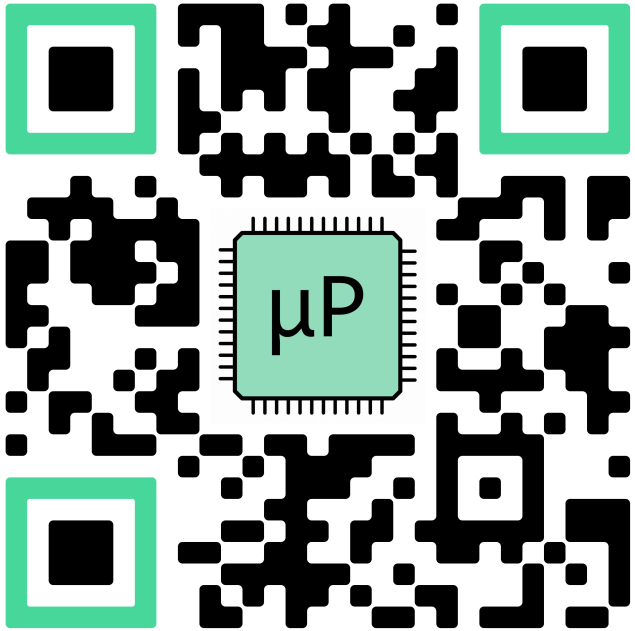
LD3 PWR: the red LED indicates that the STM32 part is powered and +5V power is available.



Feedback

- What is the most important thing you learned in class today?
- What point was most unclear from lecture today?

https://pollev.com/joshbrake155



Summary

- I will hold office hours for this week tomorrow afternoon on Zoom 1:30—2:30 (not permanent time).
- Lab demo of ADALM2000 and Scopy after this!