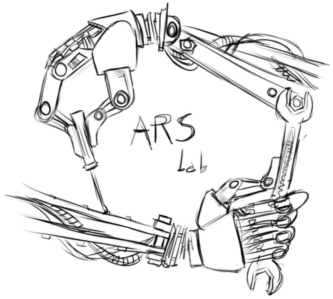
# The 7-segments Display Driver

Corrado Santoro

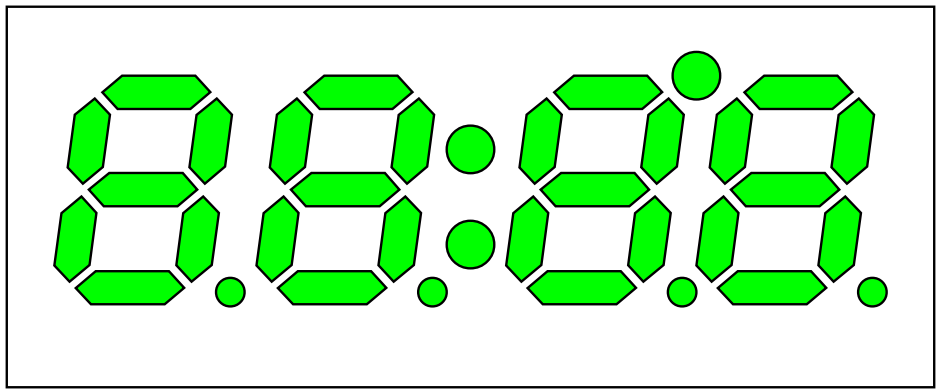
**ARSLAB - Autonomous and Robotic Systems Laboratory**
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it




L.S.M. Course

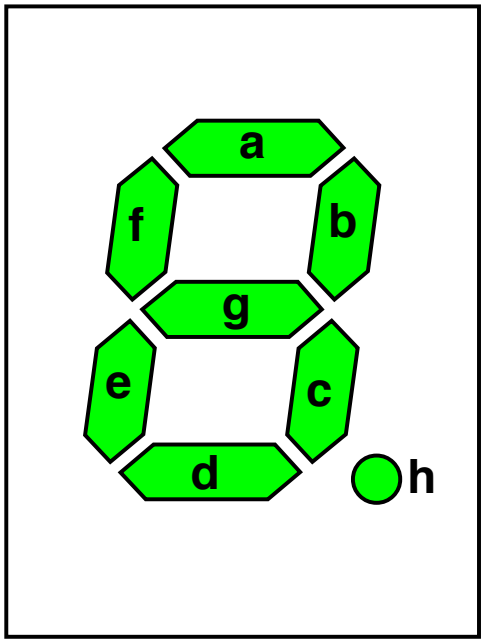
# The 7-segments Display

- The Nucleo64 add-on board has a **four digit** 7-segments display
- Each digit is made of 8 LEDs (7-segments + decimal dot)
- In theory, each LED can be lit using a digital output ...
- ... but the number of outputs needed would be too high $\rightarrow$ **8 LEDs * 4 digits = 32 digital outputs!!!**



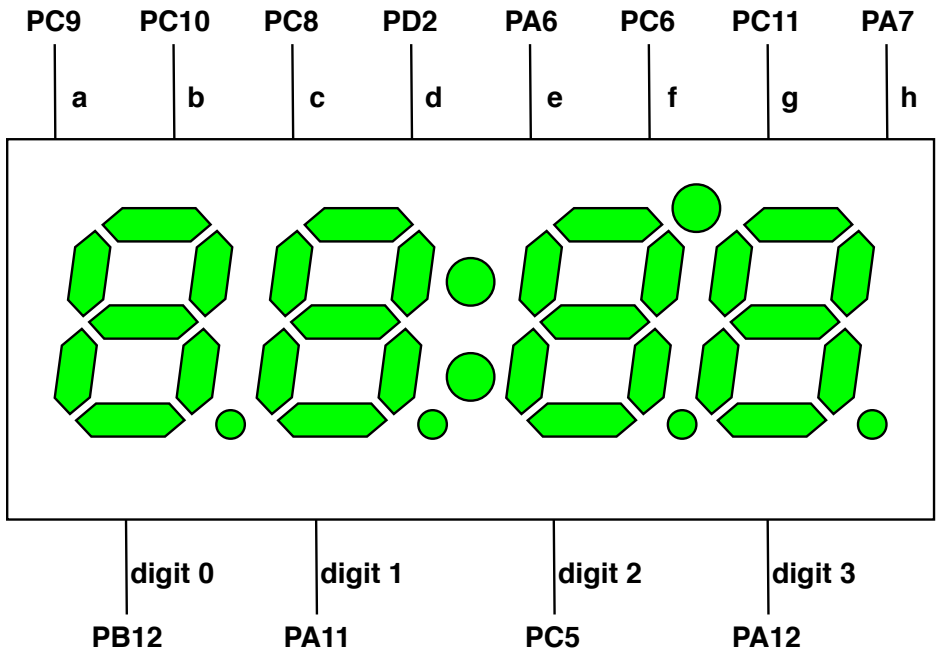
# Naming Standard for 7-segments Display

- In a 7-segments display, all the segments of each digit are named using alphabet letters **a, b, c, d, e, f, g, h**



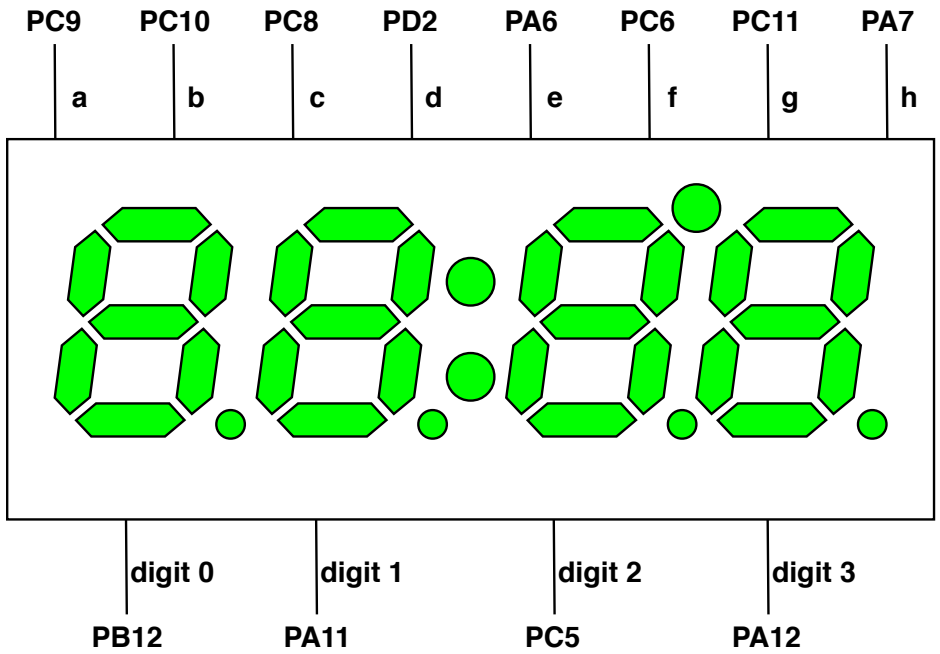
# 4-Digit / 7-Segments Display Connection

- The display used in the Nucleo board is organised in such a way as to avoid to use **too much** digital output lines
- It has a single connection per **segment name** and a single connection per **digit**



# 4-Digit / 7-Segments Display Connection

- In order to show something on a **specific segment**, we must:
  - activate the lines relevant to **segments** to be lit
  - activate the lines relevant to the **digit** to use
- at first sight, this mode **impedes** having different segments lit in different digits



# Example: let's write "0" in digits 0 and 3

```
#include "stm32_unict_lib.h"
#include <stdio.h>

int counter = 0;

int main()
{
    GPIO_init(GPIOA);
    GPIO_init(GPIOB);
    GPIO_init(GPIOC);
    GPIO_init(GPIOD);
    GPIO_config_output(GPIOC, 9); // a
    GPIO_config_output(GPIOC, 10); // b
    ...
    GPIO_config_output(GPIOA, 12); // digit 3

    // write ``0''
    GPIO_write(GPIOC,  9, 1); // ``a'' ON
    GPIO_write(GPIOC, 10, 1); // ``b'' ON
    GPIO_write(GPIOC,  8, 1); // ``c'' ON
    GPIO_write(GPIOD,  2, 1); // ``d'' ON
    GPIO_write(GPIOA,  6, 1); // ``e'' ON
    GPIO_write(GPIOC,  6, 1); // ``f'' ON
    GPIO_write(GPIOC, 11, 0); // ``g'' OFF
    GPIO_write(GPIOA,  7, 0); // ``h'' OFF

    GPIO_write(GPIOB, 12, 1); // digit 0 ON
    GPIO_write(GPIOA, 11, 0); // digit 1 OFF
    GPIO_write(GPIOC,  5, 0); // digit 2 OFF
    GPIO_write(GPIOA, 12, 1); // digit 3 ON
    for (;;) {}
}
```

# Mapping Characters to Segments

- To show characters on the display, we must **map** the ASCII code of each character to a **bit pattern** that represents the segments to lit
- Let's represent **segments** according to the following bit pattern:

| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | h |

- For example, to show a "1" digit, we must lit segments **b** and **c**:

| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | = 0x60 |

# Mapping Characters to Segments

- Mapping can be performed by using an array of **`uint8_t`** of 256 elements:
  - the ASCII code is used as the index of the array
  - the indexed element contains the relevant bit pattern

```
uint8_t character_table[256];

...
character_table['0'] = 0xfc; // 1111 1100
character_table['1'] = 0x60; // 0110 0000
character_table['2'] = 0xda; // 1101 1010
character_table['3'] = 0xf2; // 1111 0010
character_table['4'] = 0x66; // 0110 0110
character_table['5'] = 0xb6; // 1011 0110
character_table['6'] = 0xbe; // 1011 1110
character_table['7'] = 0xe0; // 1110 0000
character_table['8'] = 0xfe; // 1111 1110
character_table['9'] = 0xf6; // 1111 0110
character_table['='] = 0x12; // 0001 0010
character_table['A'] = 0xee; // 1110 1110
character_table['a'] = 0xfa; // 1011 1110
character_table['B'] = 0x3e; // 0011 1110
character_table['b'] = 0x3e; // 0011 1110
character_table['C'] = 0x9c; // 1001 1100
character_table['c'] = 0x1a; // 0001 1010
...
```

# Mapping Characters to Segments

- The next step implies to analyse the bit pattern and writing proper values to GPIO lines relevant to segments ...

```
void display_putc(uint8_t c)
{
  uint8_t bit_pattern = character_table[c];
  // a
  if ((bit_pattern & 0x80) != 0) GPIO_Write(GPIOC, 9, 1) else GPIO_Write(GPIOC, 9, 0);
  // b
  if ((bit_pattern & 0x40) != 0) GPIO_Write(GPIOC,10, 1) else GPIO_Write(GPIOC,10, 0);
  // c
  if ((bit_pattern & 0x20) != 0) GPIO_Write(GPIOC, 8, 1) else GPIO_Write(GPIOC, 8, 0);
  // d
  if ((bit_pattern & 0x10) != 0) GPIO_Write(GPIOD, 2, 1) else GPIO_Write(GPIOD, 2, 0);
  // e
  if ((bit_pattern & 0x08) != 0) GPIO_Write(GPIOA, 6, 1) else GPIO_Write(GPIOA, 6, 0);
  // f
  if ((bit_pattern & 0x04) != 0) GPIO_Write(GPIOC, 6, 1) else GPIO_Write(GPIOC, 6, 0);
  // g
  if ((bit_pattern & 0x02) != 0) GPIO_Write(GPIOC,11, 1) else GPIO_Write(GPIOC,11, 0);
  // h
  if ((bit_pattern & 0x01) != 0) GPIO_Write(GPIOA, 7, 1) else GPIO_Write(GPIOA, 7, 0);
}
```

# Mapping Characters to Segments

- ... and then to “turn-on” the proper digit

```
void digit_on(int digit, int on_off)
{
  switch (digit) {
  case 0:
    GPIO_write(GPIOB, 12, on_off);
    GPIO_write(GPIOA, 11, 0);
    GPIO_write(GPIOC,  5, 0);
    GPIO_write(GPIOA, 12, 0);
    break;
  case 1:
    GPIO_write(GPIOB, 12, 0);
    GPIO_write(GPIOA, 11, on_off);
    GPIO_write(GPIOC,  5, 0);
    GPIO_write(GPIOA, 12, 0);
    break;
  case 2:
    GPIO_write(GPIOB, 12, 0);
    GPIO_write(GPIOA, 11, 0);
    GPIO_write(GPIOC,  5, on_off);
    GPIO_write(GPIOA, 12, 0);
    break;
  case 3:
    GPIO_write(GPIOB, 12, 0);
    GPIO_write(GPIOA, 11, 0);
    GPIO_write(GPIOC,  5, 0);
    GPIO_write(GPIOA, 12, on_off);
    break;
  }
}
```

# Parametric Implementation

- The listing shows that each segment is associated to:
  - A bit pattern: 0x80, 0x40, ...
  - A GPIO port: GPIOA, GPIOB, ...
  - A GPIO pin: 9, 10, 8, ...

```
void display_putc(uint8_t c)
{
  uint8_t bit_pattern = character_table[c];
  // a
  if ((bit_pattern & 0x80) != 0) GPIO_Write(GPIOC, 9, 1) else GPIO_Write(GPIOC, 9, 0);
  // b
  if ((bit_pattern & 0x40) != 0) GPIO_Write(GPIOC,10, 1) else GPIO_Write(GPIOC,10, 0);
  // c
  if ((bit_pattern & 0x20) != 0) GPIO_Write(GPIOC, 8, 1) else GPIO_Write(GPIOC, 8, 0);
  // d
  if ((bit_pattern & 0x10) != 0) GPIO_Write(GPIOD, 2, 1) else GPIO_Write(GPIOD, 2, 0);
  // e
  if ((bit_pattern & 0x08) != 0) GPIO_Write(GPIOA, 6, 1) else GPIO_Write(GPIOA, 6, 0);
  // f
  if ((bit_pattern & 0x04) != 0) GPIO_Write(GPIOC, 6, 1) else GPIO_Write(GPIOC, 6, 0);
  // g
  if ((bit_pattern & 0x02) != 0) GPIO_Write(GPIOC,11, 1) else GPIO_Write(GPIOC,11, 0);
  // h
  if ((bit_pattern & 0x01) != 0) GPIO_Write(GPIOA, 7, 1) else GPIO_Write(GPIOA, 7, 0);
}
```

# Parametric Implementation

- The listing shows that each segment is associated to:
  - A bit pattern: 0x80, 0x40, ...
  - A GPIO port: GPIOA, GPIOB, ...
  - A GPIO pin: 9, 10, 8, ...
- We define a **structure** representing a segment and an array variable representing all the segments.

```
typedef struct {
    uint8_t pattern;
    GPIO_TypeDef * port;
    int pin;
} t_segment;

t_segment segment_table[8] = {
    { 0x80, GPIOC, 9},  // segment "A"
    { 0x40, GPIOC, 10}, // segment "B"
    { 0x20, GPIOC, 8},  // segment "C"
    { 0x10, GPIOD, 2},  // segment "D"
    { 0x08, GPIOA, 6},  // segment "E"
    { 0x04, GPIOC, 6},  // segment "F"
    { 0x02, GPIOC, 11}, // segment "G"
    { 0x01, GPIOA, 7},  // segment "H"
};
```

# Parametric Implementation

- The function is reduced to a loop:

```
typedef struct {
    uint8_t pattern;
    GPIO_TypeDef * port;
    int pin;
} t_segment;

t_segment segment_table[8] = {
    { 0x80, GPIOC, 9},  // segment "A"
    { 0x40, GPIOC, 10}, // segment "B"
    { 0x20, GPIOC, 8},  // segment "C"
    { 0x10, GPIOD, 2},  // segment "D"
    { 0x08, GPIOA, 6},  // segment "E"
    { 0x04, GPIOC, 6},  // segment "F"
    { 0x02, GPIOC, 11}, // segment "G"
    { 0x01, GPIOA, 7},  // segment "H"
};

void display_putc(uint8_t c)
{
    int i;
    uint8_t bit_pattern = character_table[c];
    for (i = 0;i < 8;i++) {
        t_segment * s = &segment_table[i];
        if ((bit_pattern & s->pattern) != 0)
            GPIO_write(s->port,s->pin, 1);
        else
            GPIO_write(s->port,s->pin, 0);
    }
}
```

# Multiplexing

- According to the way in which the display is interfaced, we **cannot** lit different segments in different digits, unless ...
- ... we handle a digit at time, using a time-sharing approach:
  - We define a timer generating an interrupt at a high frequency, $< 1ms$
  - At each interrupt, we turn off the current digit and turn on the next digit (after having prepared the proper segments to lit)
- To this aim, we must use a **four-elements buffer** to hold the characters to be displayed in each digit
- User program writes characters in the buffer, and the interrupt handler takes care of updating the display

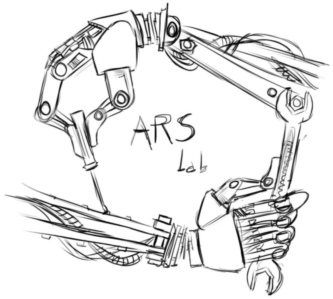
# Muliplexing

```
int current_scheduled_digit = 0; // the current digit lit
char current_digit_pattern[4] = { 0, 0, 0, 0}; // the buffer

void DISPLAY_putc(int digit, char c)
{
    current_digit_pattern[digit] = character_table[(int)c];
}

void TIM5_IRQHandler(void)
{
    if (TIM_update_check(TIM5)) {
        digit_on_off(current_scheduled_digit, 0);
        current_scheduled_digit = (current_scheduled_digit + 1) % 4;
        DISPLAY_set(current_digit_pattern[current_scheduled_digit]);
        digit_on_off(current_scheduled_digit, 1);
        TIM_update_clear(TIM5);
    }
}
```

# The 7-segments Display Driver

Corrado Santoro

**ARSLAB - Autonomous and Robotic Systems Laboratory**
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it




L.S.M. Course