Understanding the STM32F0's GPIO part 1 (./stm32f0-gpio-tutorial-part-1.html)

This is the first part of the GPIO tutorial for the STM32F0Discovery. The tutorial will include the following topics:

- Understanding the Structure of the GPIO Registers on the STM32F0 Microcontroller
- Understanding how to access the registers
- Setting and clearing output pins

Part 2 (./stm32f0-gpio-tutorial-part-2.html) will cover:

- RGB LED control
- Reading from inputs pins
- Hardware considerations

The STM32F051 microcontroller on the STM32F0Discovery board has 5 general purpose input/output (GPIO) ports named Port A, B, C, D and F. Each port can have up to 16 pins, and each port has associated with it the following set of registers:

- GPIO port mode register (GPIOx_MODER)
- GPIO port output type register (GPIOx_OTYPER)
- GPIO port output speed register (GPIOx_OSPEEDR)
- GPIO port pull-up/pull-down register (GPIOx_PUPDR)
- GPIO port input data register (GPIOx _IDR)
- GPIO port outp ut data register (GPIOx_ODR)
- GPIO port bit set/reset register (GPIOx _BSRR)
- GPIO port configuration lock register (GPIOx_LCKR)
- GPIO alternate function low register (GPI Ox_AFRL)
- GPIO alternate function high register (GPIOx_AFRH)
- GPIO Port bit reset register (GPIOx_BRR)

where the 'x' in each register name acronym represents the port i.e. the GPIOx_MODER associated with port A is called GPIOA_MODER. Let's take a closer look at each register:

GPIO port mode register (GPIOx_MODER)

This is a 32-bit register where each set of two consecutive bits represent the mode of a single I/O pin. For example bits 0 and 1 of the MODER register associated with GPIOC (GPIOC_MODER), represent the mode of GPIO pin PC0 and bits 26 and 27 of the same register represent the mode of GPIO pin PC13. These two bits can be set to:

- '00'-> input mode, which allows the GPIO pin to be used as an input pin,
- '01'-> Output mode, which allows the GPIO pin to be used as an output pin,
- '11'-> Analog mode, which allows the GPIO pin to be used as an Analog input pin and finally,
- '10'-> Alternate function mode which allow the GPIO pins to be used by peripherals such as the UART, SPI e.t.c. It is important to note that if a pin's MODE is set to alternate function, any GPIO settings for that pin in the GPIO registers will be overridden by the peripheral. I will be addressing Alternate function mode in more detail in a future entry.

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+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

(files/uploads/2012/07/art4image11.png)
Figure 1 - GPIOx_MODER register

GPIO port output type register (GPIOx_OTYPER)

This is a 16-bit register where each bit denotes the 'type' of a single pin in the register. This register sets the type of output pins to either push-pull or open drain. For example if pin PC7 is configured as an output pin, clearing bit 7 (or leaving its state at zero) of the OTYPER register associated with GPIOC (GPIOC_TYPER), will set the output type of the GPIO output pin PC7 to "Push-Pull".

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
	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
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 **OTy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain

(files/uploads/2012/07/art4image21.png)
Figure 2. GPIOx_OTYPER Register

GPIO port output speed register (GPIOx_OSPEEDR)

This is a 32-bit register where each set of two bits represent the speed of a single output pin. For example bits 0 and 1 of the OSPEEDR register associated with port C (GPIOC_OSPEEDR), represent the speed setting of the output pin PC0 and bits 26 and 27 of the same register represent the speed setting of the output pin PC13. These two bits can be set to:

- 'x0': 2MHz Low speed
- '01':10MHz Medium speed
- '11': 50MHz High speed

So why have a speed setting on I/O ? To save power. On the 2MHz setting the GPIO would consume less current than on the 50MHz setting I'd imagine but would have relatively longer rise/fall time specs.

The User Manual for the STM32F0 claims that the the output pins fastest toggle speed is every two clock cycles. Assuming a maximum operation speed of 48MHz, the fastest toggle speed for the GPIO on the STM32f0 is 24 MHZ, which means the highest frequency square wave that can be produced by the GPIO is 12MHz.

NOTE: A quick look at the default startup code found in"system_stm32f0xx.c" will identify that the microcontroller is indeed operating at a maximum speed of 48MHz at startup. In a future entry I will demonstrate how this speed can be changed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[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
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **OSPEEDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output speed.
x0: 2 MHz Low speed
01: 10 MHz Medium speed
11: 50 MHz High speed

(files/uploads/2012/07/art4image41.png)
Figure 3. GPIOx_OSPEEDR register

GPIO port pull-up/pull-down register (GPIOx_PUPDR)

The GPIOx_PUPDR registers configures the internal pull-ups and pull-down resistors on each I/O pin. The internal pull-up/down resistors can be configured on GPIO pins set as input or output (though I'd imagine they'd be more popular on input pins). The Pullup/down resistors have a typical value of 40KOhms but can range from 30-50Kohms.

Again each two consecutive bits represent the internal pull-up/down resistor setting for each pin within a single port.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

(files/uploads/2012/07/art4image51.png)
Figure 4. GPIOx_PUPDR Register

GPIO port input data register (GPIOx _IDR)

This is a 16-bit read-only register. Each bit represents the input value on a corresponding pin. Reading a '0' in bit 8 of this GPIOC _IDR register indicates that the voltage on PC8 is 0V (GND). While reading a '1' in bit 8 of this GPIOC _IDR register indicates that the voltage on PC8 is 3.3V (VDD)

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

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

Bits 15:0 **IDR[15:0]:** Port input data

These bits are read-only. They contain the input value of the corresponding I/O port.

(files/uploads/2012/07/art4image61.png) Figure 5. GPIOx_IDR Register

GPIO port output data register (GPIOx_ODR)

This is a 16-bit read/write register. Each bit represents the output value on a corresponding pin. Writing a '0' in bit 8 of this GPIOC _ODR register indicates that the voltage on PC8 is driven by the micro to 0V (GND). While writing a '1' in bit 8 of this GPIOC _ODR register indicates that the voltage on PC8 is driven by the micro to 3.3V (VDD).

Writing to the ODR register is good if you want to write to the entire port.e.g.

```
GPIOC->ODR = 0xF0FE
```

The above statement changes the state of every pin on the GPIOC peripheral from its previous (and now discarded) state, to the one indicated by the statement; 0xF0FE (0b1111000011111110).

However if you want to set only a single pin; lets say PC8 without affecting the state of the rest of the pins on GPIOC, you have to perform a read-modify-write (RMW) access. To set pin PC8 independent of all other pins on GPIOC (RMW) you could use:

```
GPIOC->ODR |=  0x00000100;  //( 0b00000000000000000000000100000000)
```

To clear pin PC8 independent of all other pins on GPIOC (RMW) you could use:

```
GPIOC->ODR &=  ~(0x00000100);  //( 0b00000000000000000000000100000000)
```

This works just fine, but you have to**read** the ODR register, OR (|) or AND(&) **(modify)** it with a mask and then **write** it back to the ODR register. This means that at the assembly language level, at least three instructions are used to set/clear an I/O which can significantly slow down toggling speed. A better way would be to use the BSRR register and the BRR registers for setting and clearing pins. They enable 'atomic' access that allows the I/O pin to be clear/set in as short a time as possible.

GPIO port bit set/reset register (GPIOx _BSRR)

As mentioned in the preceding paragraph the BSRR register allows us to set/clear a particular pin (or groups of pins) while preserving the state of the rest of the pins on a GPIO peripheral atomically i.e. a fast as possible, without having to resort to the slower read-modify-write (RMW) accesses. The least significant 16 bits are used to atomically set pin values to VDD whereas the most significant 16 bits are used to atomically clear pin values to GND.

So if I wanted to set PC8 independent of all other pins on GPIOC I could use:

```
GPIOC->BSRR = 0x00000100;//( 0b00000000000000000000000100000000)
```

or

```
GPIOC->BSRR = (1<<8);
```

To clear pin PC8 independent of all other pins on GPIOC you could use:

```
GPIOC->BSRR = 0x01000000; //( 0b00000001000000000000000000000000)
```

or

```
GPIOC->BSRR = (1<<24);
```

Notice how in both scenarios a simple assignment operator '=' (atomic) was used rather than an '|=' or an '&=' which denote RMW accesses. Furthermore, note that to clear the pin value of PC8 to GND, I had to set the 24th bit in the BSRR (8th bit of the most significant 16 bits). While to set the pin value of PC8 to VDD, I had to set the 8th bit in the BSRR. The awkwardness of atomic clearing being mapped to the most significant 16-bits of the BSRR register is compensated for by the inclusion of the BRR register. The BRR register maps the most significant 16-bits of the BSRR register into itself. So to clear pin PC8 independent of all other pins on GPIOC you could use:

```
GPIOC->BRR = (1<<8);
```

Finally there are three more registers; the GPIOx_AFRH, GPIOx_AFRL, and the GPIOx_LCKR registers. The first two allow GPIO pins to be used for alternate functions. There is a really neat pin muxing mechanism that allows each GPIO to be mapped to multiple alternate functions depending on how these two registers are set. I will spend more time on the AFRL/AFRH registers in future entries. The last GPIOx_LCKR register can be used once GPIO is configured to 'lock' the configuration so that it does not change until the micro is reset. I encourage you to look up these three registers in the user manual.

Accessing Registers in C

The next step will be to look at how we access registers and perform bit manipulation. For setting bits in a register we can use the following:

```
PERIPHERAL->REG |= (1 << 2) ;
```

This will set bit 2 in register 'REG' in peripheral 'PERIPHERAL'. Alternately we could also specify:

```
PERIPHERAL->REG |=0x04; //hex
PERIPHERAL->REG |=4; //decimal
PERIPHERAL->REG |=0b100;//binary
```

This statement allows as to OR (|) the value of REG with a 'mask' that is "0b00000000000000000000000000000100" and store the result back in REG . This will have the effect of setting bit 2 in the REG register while preserving the state of all the other bits in REG.

Now to clear bit 2 in register 'REG' in peripheral 'PERIPHERAL'.

```
PERIPHERAL->REG &= ~(1 << 2) ;
```

This statement allows as to AND (&) the value of REG with a 'mask' that is the complement of "0b00000000000000000000000000000100" (which is "11111111111111111111111111111011") and store the result back in REG . This will have the effect of clearing bit 2 in the REG register while preserving the state of all the other bits in REG.

Notice that all of these accesses are RMW. One can use RMW access to configure registers. But when it comes to toggling I/O, especially when speed is a concern, one should use atomic access via the BSRR/BRR registers.

But what about all of this '->' business ? What does this mean? In general in the C programming language PERIPHERAL->REG means accessing "variable REG in a structure pointed to by pointer PERIPHERAL". I'll try and explain this further via the "stm32f0xx.h" header file. This file can be found in "C:\Development\STM32F0-

Discovery_FW_V1.0.0\Libraries\CMSIS\ST\STM32F0xx\Include" if you followed the instructions in the tutorials.

On line 396 of the "stm32f0xx.h" header file, you'll see a structure definition for the GPIO peripheral.

```
typedef struct //396
{
  __IO uint32_t MODER;        /*!< GPIO port mode register,
    Address offset: 0x00 */
  __IO uint16_t OTYPER;       /*!< GPIO port output type register,
    Address offset: 0x04 */
  uint16_t RESERVED0;         /*!< Reserved,
                   0x06 */
  __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 uint16_t IDR;          /*!< GPIO port input data register,
    Address offset: 0x10 */
  uint16_t RESERVED1;         /*!< Reserved,
                   0x12 */
  __IO uint16_t ODR;          /*!< GPIO port output data register,
    Address offset: 0x14 */
  uint16_t RESERVED2;         /*!< Reserved,
                   0x16 */
  __IO uint32_t BSRR;         /*!< GPIO port bit set/reset registerBSRR,
    Address offset: 0x18 */
  __IO uint32_t LCKR;         /*!< GPIO port configuration lock register,
    Address offset: 0x1C */
  __IO uint32_t AFR[2];       /*!< GPIO alternate function low register,               A
ddress offset: 0x20-0x24 */
  __IO uint16_t BRR;          /*!< GPIO bit reset register,
    Address offset: 0x28 */
  uint16_t RESERVED3;         /*!< Reserved,
                   0x2A */
}GPIO_TypeDef;
```

At line 708 we see define statements that define the base address of each GPIO peripheral GPIOA, GPIOB, GPIOC, GPIOD and GPIOF.

```
//line 708
#define GPIOA_BASE            (AHB2PERIPH_BASE + 0x00000000)
#define GPIOB_BASE            (AHB2PERIPH_BASE + 0x00000400)
#define GPIOC_BASE            (AHB2PERIPH_BASE + 0x00000800)
#define GPIOD_BASE            (AHB2PERIPH_BASE + 0x00000C00)
#define GPIOF_BASE            (AHB2PERIPH_BASE + 0x00001400)
```

Now finally on line 762 we see that the base address of each peripheral is 'cast' as a pointer to the GPIO peripheral structure.

This has the effect of mapping a pointer to the GPIO structure "(GPIO_TypeDef *)", at the respective GPIO peripheral base address...say "GPIOA_BASE" and that this pointer can be accessed by its alias "GPIOA".

```
//line 762
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC                ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOF                ((GPIO_TypeDef *) GPIOF_BASE)
```

But we're not done yet! Further down in the same header file we'll see all types of bit definitions that can replace the "(1 <<2)" explained above with something more useful. For example lets say that we want to set the mode of pin PC8 to output. We could do that via:

```
GPIOC-> MODER |= (1 << 16);
```

Alternatively we could use:

```
GPIOC-> MODER |= GPIO_MODER_MODER8_0 ;
```

Why? because at line 1446 of the header file we find the following define statement

```
#define GPIO_MODER_MODER8_0         ((uint32_t)0x00010000)
```

Using the second option makes quite a bit more sense than "(1<<16)" since it informs the programmer that we're setting the mode of pin PC8. This is not super informative and kind of long, but its still more informative than a simple (1 << 16).

Now what if we want to set the MODE of PC8 and PC9 to output in the same statement? we can use the bitwise OR operator to append as many flags as we like:

```
GPIOC-> MODER |= (GPIO_MODER_MODER8_0|GPIO_MODER_MODER9_0) ;
```

Now we will tackle our first complete example. This first example will have identical behavior to the iotogglem0 (files/OTHERFILES/iotogglem0.zip) example but will be coded without the use of the STM32F0 peripheral library. It will however rely heavily on use of the register definitions and bit definitions found in the "stm32f0xx.h" header file. This will not only make our code size significantly smaller, it will also mean that the makefile will not be as complicated and the build process will not depend on the location of the peripheral library on your hard drive i.e. its self-contained. The project folder for this project is called gpiotogglep1 and be downloaded from here (files/OTHERFILES/gpiotogglep1.zip) .

The complete main.c source file is shown below:

```
#include "stm32f0xx.h"

#define BSRR_VAL        0x0300

void delay (int a);

int main(void)
{
  /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f0xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f0xx.c file
     */

  /* GPIOC Periph clock enable */
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    GPIOC->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0) ;
    /* Configure PC8 and PC9 in output  mode  */

    GPIOC->OTYPER &= ~(GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9) ;
    // Ensure push pull mode selected--default

    GPIOC->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR8|GPIO_OSPEEDER_OSPEEDR9);
    //Ensure maximum speed setting (even though it is unnecessary)

    GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR8|GPIO_PUPDR_PUPDR9);
    //Ensure all pull up pull down resistors are disabled

    while (1)
    {
        /* Set PC8 and PC9 */
        GPIOC->BSRR = BSRR_VAL;
        delay(500000);
        /* Reset PC8 and PC9 */
        GPIOC->BRR = BSRR_VAL;
        delay(500000);
    }

    return 0;
}

void delay (int a)
{
    volatile int i,j;

    for (i=0 ; i < a ; i++)
    {
        j++;
    }
```

```
    return;
}
```

The first three lines include an include statement for the "stm32f0xx.h" header file, a define statement for a value that will be used to toggle PC8 and PC9, which are connected to the LEDs on the the STM32F0 Discovery Board, and finally a simple delay function protoype.

```
/* GPIOC Periph clock enable */
  RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
```

The first statement in the main function is shown above. Each peripheral on the the STM32F0 microcontroller is clock gated. This means that the clock signal does not reach the peripheral until we tell it to do so by way of setting a bit in a register. This means that the clock signal never reaches peripherals that we do not use and thus saves power. To enable the clock to reach the GPIOC peripheral, we need to set the "IOPCEN" bit (bit 19) in the AHBENR register in the RCC (Reset and Clock Control)peripheral. At this point I encourage you to take a look at the RCC peripheral in the user manual and to identify the bits that "gate" the clock for other peripherals such as the other GPIO peripherals (A,B,e.t.c), the SPI, USART e.t.c. I promise you that you'll find one bit per peripheral used for clock gating.

Once we enable the clock to the GPIOC peripheral, we can configure the GPIOC peripheral as shown below. The first line configures PC8 and PC9 as outputs, the second line ensures that both pins are in push-pull mode, the third line ensures that the maximum speed setting is set for both pins and the last line ensures that pullups/pull downs are disabled on both pins. At this point I strongly encourage you to look up the bit definitions on the left hand side of each statement up in the "stm32f0xx.h" header file to confirm that these statements are indeed correct.

```
GPIOC->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0) ;
/* Configure PC8 and PC9 in output  mode  */

GPIOC->OTYPER &= ~(GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9) ;
// Ensure push pull mode selected--default

GPIOC->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR8|GPIO_OSPEEDER_OSPEEDR9);
//Ensure maximum speed setting (even though it is unnecessary)

GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR8|GPIO_PUPDR_PUPDR9);
//Ensure all pull up pull down resistors are disabled
```

The next block of code is a simple infinite (super) loop that writes bits 8 and 9 in the GPIOC_BSRR register to atomically set PC8 and PC9 pins to VDD followed by a delay and then followed by a write to bits 8 and 9 in the GPIOC_BRR register to atomically clear PC8 and PC9 pins to GND.

```
while (1)
   {
       /* Set PC8 and PC9 */
       GPIOC->BSRR = BSRR_VAL;
       delay(500000);
       /* Reset PC8 and PC9 */
       GPIOC->BRR = BSRR_VAL;
       delay(500000);
   }
```

The program size for the non-peripheral library based gpiotogglep1 project is 864 bytes, whereas the program size for the peripheral library based iotogglemo project that performs the same function was 2556 bytes! that's an almost 3x program memory saving!

In all fairness however this is a very small example. A fairer comparison could be achieved with a significantly larger project. But it does demonstrate that the peripheral library usage will consume more Flash memory.

- « The iotogglem0 Project Folder Structure and the makefile (./the-iotogglem0-project-folder-structure-and-the-makefile.html)
- Understanding the STM32F0′s GPIO part 2 (./stm32f0-gpio-tutorial-part-2.html) »

Published

Jul 28, 2012

Category

STM32F0 (/categories.html#STM32F0-ref)

Tags

- GPIO 7 (/tags.html#GPIO-ref)
- STM32F0 11 (/tags.html#STM32F0-ref)