**GPIO Programming: Interfacing Input and Output Devices**

**1. Aim:**

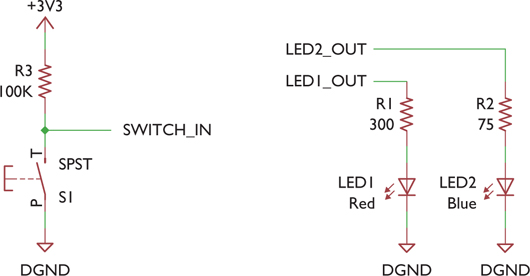
Write C programs to communicate with basic devices such as switches and light-emitting diodes (LEDs) using General Purpose Input /Output (GPIO) ports of STM32F407 microcontroller.

**2. Introduction:**

Embedded computers typically need to sense their environment and then control devices in response. Let’s start with the basics by learning how to make an embedded computer flash light and read a switch by using a general purpose input/output (GPIO) port. A microcontroller unit (MCU) contains a central processing unit (CPU) that executes instructions. The CPU is surrounded by specialized hardware circuits called peripherals that interface with external devices or perform other functions for the MCU. Some of these peripherals are integrated into the MCU, whereas others may be added externally on the printed circuit board.

An input GPIO port bit lets us read a single bit digital value on an MCU pin. If we connect the pin to a switch (as with the signal SWITCH\_IN), then the program can tell if the switch is open or closed. An output GPIO port bit enables the program to set an MCU pin to one of two voltage levels (e.g., either 3.3 V or 0 V). A program can light or extinguish an LED that is connected to this output pin (e.g., LED1\_OUT or LED2\_OUT).

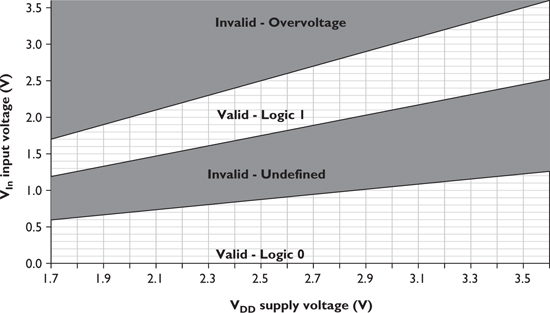
**Basic digital input and output circuit:**



An MCU is a digital computer, so it operates on ones and zeros. These values are represented by voltages within specific ranges relative to the MCU’s power supply voltage, VDD. The actual voltages do not matter much to an embedded system developer when they are inside the MCU, but they do matter when we want to interface with external devices.

**2.1 Input Signals:**

Digital inputs are interpreted based on their voltage levels. The supply voltage VDD sets the thresholds for determining whether an input voltage will be considered a one or a zero. For example, the MCU’s data sheet might specify that an input voltage between 0 V and 0.35 × VDD will be interpreted as a logic zero, whereas an input voltage between 0.7 × VDD and VDD will be interpreted as a logic one. Figure shows how input voltages are interpreted based on the supply voltage VDD.



For example, if VDD is 3.3 V, then input values between 0 and 1.155 V will be a logic zero and those between 2.31 and 3.3 V will be a logic one. An input voltage between 0.35 × VDD and 0.7 × VDD is undefined and may be read as a one or a zero. This is useless so we design external circuits to keep the voltage out of that range except when switching. Digital inputs do not draw much current from the signal source; they have high input impedance. Typically each GPIO pin will draw no more than 1 μA over the MCU’s entire operating temperature range. At room temperature (25°C), the input current will be much smaller (e.g. no more than 25 nA).

**2.2 Output Signals:**

Output voltages for digital ports are typically specified as being within a certain voltage range from a supply rail. For common MCUs, a normal output generating a logic one will produce a voltage of VOH (out high), which is between VDD and VDD – 0.5 V. A logic zero will produce a voltage of V OL (out low) between 0 and 0.5 V.

This specification only holds true if a limited amount of current is drawn from the output. As that current increases, the output circuit’s voltage drop increases, pulling the output voltage away from the supply rails (VDD and ground). If the output circuit draws enough current, it will overload and destroy the output transistor. So be careful not to exceed the specified ratings.

For many MCUs, the output current IOH or IOL must not exceed 5 mA. Some MCUs include output drivers with more powerful transistors, allowing greater output currents. “High drive” pads may be able to source or sink up more current (e.g. up to 18 mA) and still meet the output voltage specifications. The drive current capability also depends on the supply voltage VDD. As VDD falls, the transistors have a higher output resistance, increasing the voltage drop. So at lower operating voltages, the maximum output current available falls. Lowering VDD from 3.3 to 2.0 V might cut IOH and IOL from 5 to 1.5 mA.

**3. On-board LED Blinking using Embedded C Program:**

Each LED has its anode (positive end, base of triangle) connected to an MCU output signal (either LED1\_OUT or LED2\_OUT) through a resistor. The LED’s cathode (negative end, bar) is connected to ground. An LED’s brightness is roughly proportional to the current flowing through it. This current is nearly zero for low voltages, and then rises quickly as the voltage exceeds a threshold. The exact relationship between the voltage and current depends on the type of LED and its temperature. Applying 1.6 V may not light a red LED at all, but 2.0 V will make it bright (with 20 mA of current). Raising the voltage slightly to 2.3 V brightens it more and makes the current shoot up to 43 mA. Raising it further will cause the LED to overheat and fail.

The GPIO port outputs are digital and do not offer such fine-grain voltage control. They can provide two levels: and almost ground, and almost VDD (e.g. 3.3 V). The first will correctly keep the LED off, but the second will probably burn out the LED. We need to include resistor R to limit the current through the LED and MCU driver output to a safe value.

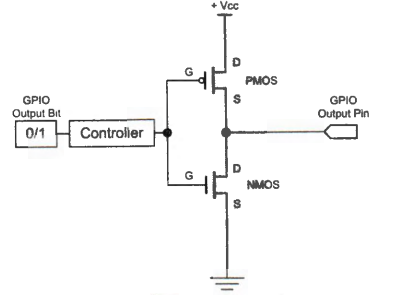
We can calculate the value R of a current-limiting resistor based on the supply voltage VDD, the LED forward voltage VF, and the desired LED current ILED:

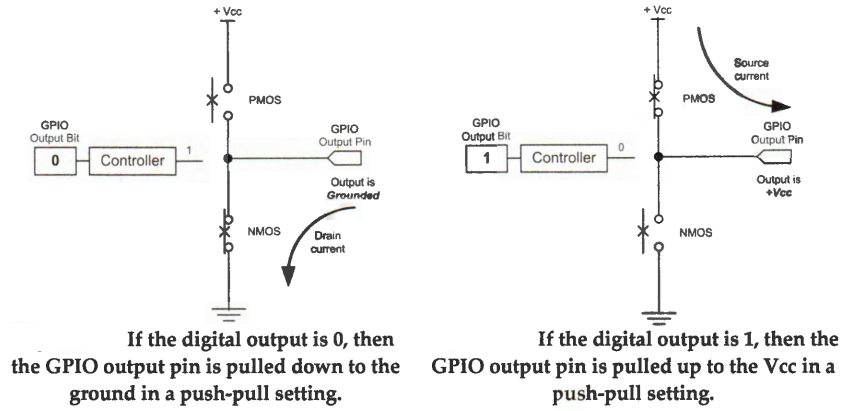
We start by picking a value of ILED which is safe for both the LED and the MCU output. In this case let’s set ILED = 4 mA, and assume VDD = 3.0 V. VF is 1.8 V for the red LED and 2.7 V for the blue LED. We can now solve for the resistor values. For the red LED, a 300 Ω resistor is needed. For the blue LED, a 75 Ω resistor is needed.

**3.1 GPIO Push-Pull Output:**

Software can configure a GPIO output pin as either push-pull or open-drain. Push-pull mode allows the pin to supply and absorb current. However, a GPIO pin in open-drain (also called collector) mode can only absorb current.

A push-pull output consists of a pair of complementary transistors, as shown in Figure and only one of them is turned on at any time.

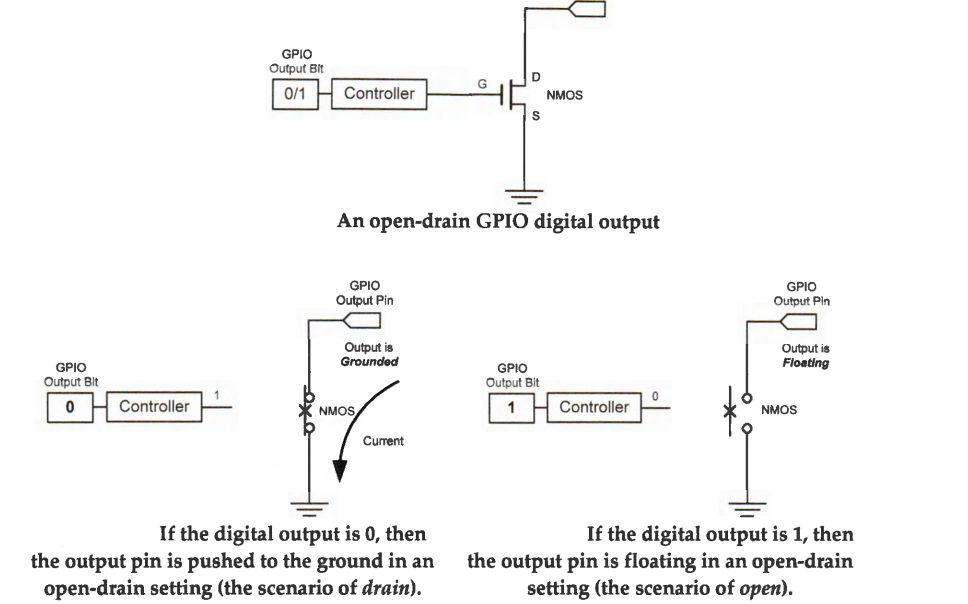




* When logic 0 is outputted, the transistor connected to the ground is turned on to sink an electric current from the external circuit, as shown in the figure above.
* When the pin outputs logic 1, the transistor connected to the power supply is turned on, and it provides an electric current to the external circuit connected to the output pin, as shown in the figure above.

**3.2 GPIO Open Drain Output:**

* When software outputs a logic 0, the open-drain circuit can sink an electric current from the external load connected to the GPIO pin.
* However, when software outputs a logic 1, it cannot supply any electric current to the external load because the output pin is floating, connected to neither the power supply nor the ground.



One important usage of open-drain outputs is to directly connect several outputs together and implement wired logic AND (active high) or OR (active low) circuit in an easy way. If multiple open-drain output pins are connected and are pulled up via a shared resistor, any output pin can drive the output voltage low. The pin voltage is high if and only if all pins output a high voltage level.

* If a high voltage level represents logic state 1 (i.e., active high), it implements a wired-AND function. The final output is 1 (high) only if all outputs of connected pins are 1 (high).
* If a low voltage level represents logic state 1 (i.e., active low), it implements a wired-OR function. The final output is 1 (low) if the output of any pins is 1 (low).

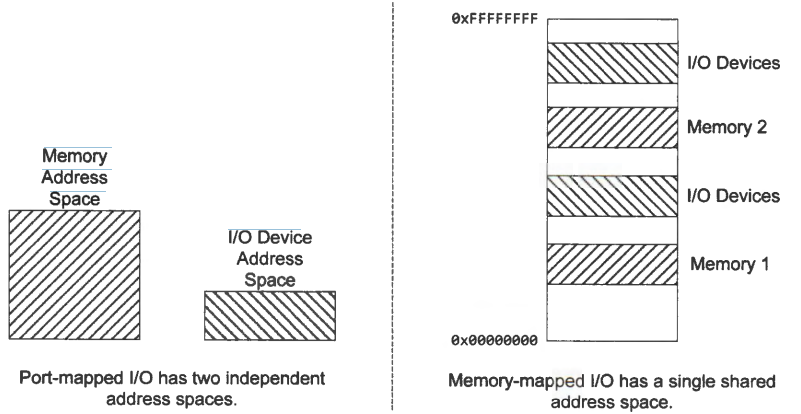
**3.3 Memory Mapped I/O:**

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 the 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 I/O, and memory-mapped I/O.

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

• Memory-mapped I/O does not need any special instructions. The memory and the 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 I/O is a more convenient way to interface I/O devices. The most significant disadvantage is that memory-mapped I/O has a more complex address decoding unit than port-mapped I/O.



ARM Cortex-M processors use memory-mapped I/O to access peripheral registers. All peripheral registers on STM32F4 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.

**4. Inside the MCU: Control Registers and C code**

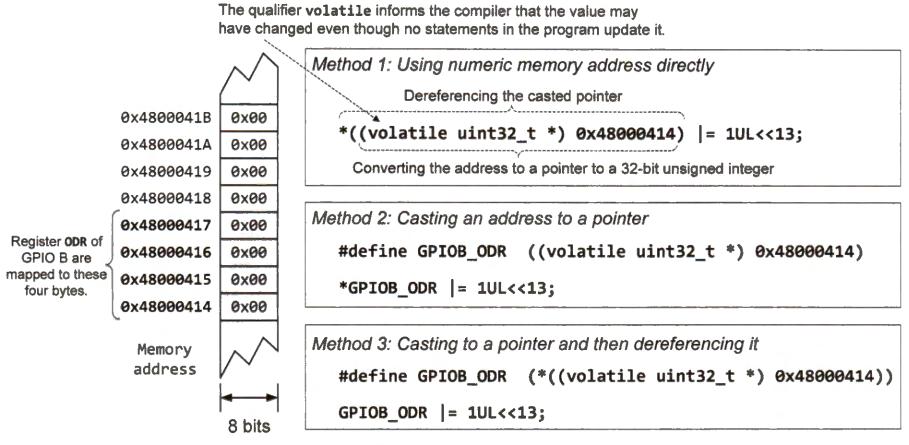
Peripherals are built with control registers to allow us to configure them, determine their status and transfer data. Software can program a GPIO pin as one of the following four different functions:

* Digital input that detects whether an external voltage signal is higher or lower than a predetermined threshold.
* Digital output that controls the voltage on the pin.
* Analog functions that perform digital-to-analog or analog-to-digital conversion.
* Other complex functions such as PWM output, LCD driver, timer-based input capture, external interrupt, and interface of USART, SPI, PC and USB communication.

A GPIO port consists of a group of GPIO pins, typically 8 or 16, which share the same data and control registers.

* When a GPIO pin i is set as a digital input, the binary data read from this pin of this GPIO group is saved at bit i in the input data register (IDR). Each bit in IDR holds the digital input of the corresponding pin.
* When a GPIO pin i is configured as a digital output, bit i in the output data register (ODR) holds the output of this pin. Therefore, when changing the output of a GPIO pin, the programmer should only alter the value of the corresponding bit of ODR, without affecting the other bits in ODR.
* All GPIO pins in a GPIO port can be configured as input or output independently.

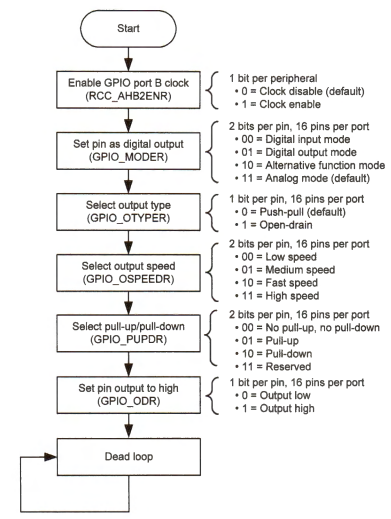
A peripheral register usually takes four bytes in memory. For example, the output data register (ODR) of Port B on STM32L4 is mapped to memory addresses 0x48000414 to 0x48000417, with the upper halfword being reserved. Note that values stored in peripheral registers are in the format of little-endian.



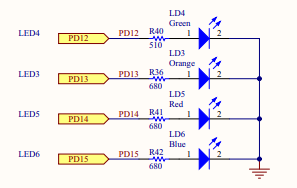
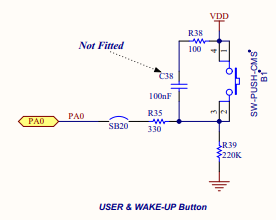
If we want to set the output of GPIO port B pin 6 to high, we can use the following C statement. lUL is an unsigned long integer with a value of 1. Note the pins are numbered 0 - 15, instead of 1 - 16.



**4.1 Program to light up a LED in STM32F407 Microcontroller:**



**4.2 Hardware Schematic Diagram:**



**4.3 LED Blinking Program:**

#include "stm32f4xx.h"

void configureLED(void);

void msDelay(volatile int msTime);

int main(void)

{

//Configure LED

configureLED();

//Define Delay function

msDelay(1000);

//GPIOD->ODR = (0x5UL<<12);

while(1)

{

GPIOD->ODR ^= (0xFUL<<12);

msDelay(250);

}

}

void configureLED(void)

{

RCC->AHB1ENR |=(1UL<<3);

GPIOD->MODER &= ~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x55UL<<12\*2);

}

void msDelay(int msTime)

{

//Assume for loop take 12 clock cycles and system clock is 16MHz

int Time=msTime\*1333;

for(int i=0;i<Time;i++);

}

**4.4 User Push Button Interface Program:**

#include "stm32f4xx.h"

volatile unsigned int PB\_state;

void configureLED(void)

{

RCC->AHB1ENR |=(1UL<<3);

GPIOD->MODER &= ~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x01UL<<12\*2);

}

void configurePB(void)

{

RCC->AHB1ENR |=(0x1UL<<0);

GPIOA->MODER &= ~(0x3UL<<0);

}

void msDelay(int msTime)

{

//Assume for loop take 12 clock cycles and system clock is 16MHz

int Time=msTime\*1333;

for(int i=0;i<Time;i++);

}

int main ()

{

configureLED();

configurePB();

while(1)

{

PB\_state=(GPIOA->IDR&(0x1UL<<0));

if(PB\_state==1)

{

GPIOD->ODR |=(0x1UL<<12);

}

else

{

GPIOD->ODR |= (0x0UL<<12);

}

}

}

**5. Conclusion:**

In this experiment, input and output activities like LED blinking and push button interfacing were performed using the GPIO ports of STM32F4 microcontroller board.