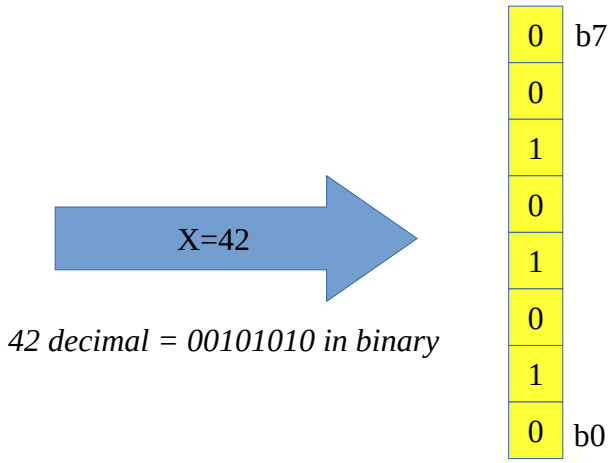
Parallel input/output

What is a parallel port?

Consider what happens in the following C code.

```
uint8_t X;
X = 42;
```

What is actually going on here? Well, the symbol X represents the contents of a 1 byte storage location somewhere in memory. When the second line is executed, the value 42 is copied in to this location. The value '42' is represented by a bit pattern (1's and 0's) which are stored in an 8 bit memory cell.



The location or address of X is determined automatically by the compiler and possibly conditions at run-time.

Certain types of memory have special properties. For example, graphics memory in a computer is often "dual-ported" [1]which means that the CPU can write to the memory over one bus (set of wires) while the GPU can read from it over a separate bus. Having two buses like this means that read and write operations can happen at the same time which speeds up things like gameplay. In the world of microcontrollers, a parallel **port** is a special kind of memory location that connects to the CPU and to pins on the side of the chip. This allows us turn devices on and off outside the chip when we write data to the port and it also allows the CPU determine the state of external devices when it reads from the port.

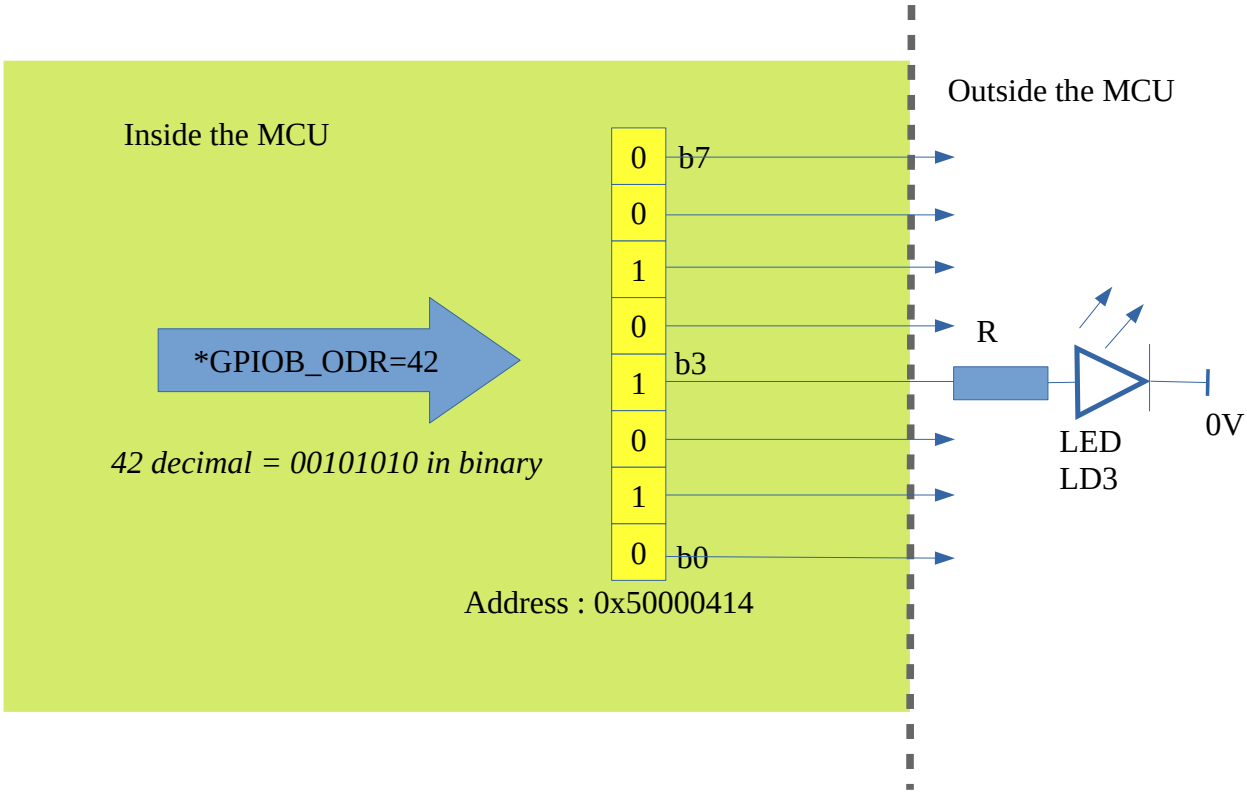
Like many microcontrollers the STM32 line of devices make use of ***memory mapped I/O***. Memory mapped I/O makes use of dual ported registers that connect to the CPU on one side and the outside world on the other.

1 A 'port' is usually thought of as a place where goods enter and leave a country. A port in the computer sense is a mechanism that allows data enter and leave a computer system.

Let's look at one of these I/O ports: **Port B**. According to the reference guide the output register for Port B is at memory location 0x50000414. In order to write to this address we need to use a pointer as follows:

```
volatile int *GPIOB_ODR = (volatile int *)0x50000414;
* GPIOB_ODR = 42;
```

Let's ignore the 'volatile' keyword for now. GPIOB_ODR is declared as being a pointer to an integer. It is told to point at memory location 0x50000414. The second line of code copies the value 42 in to this location.



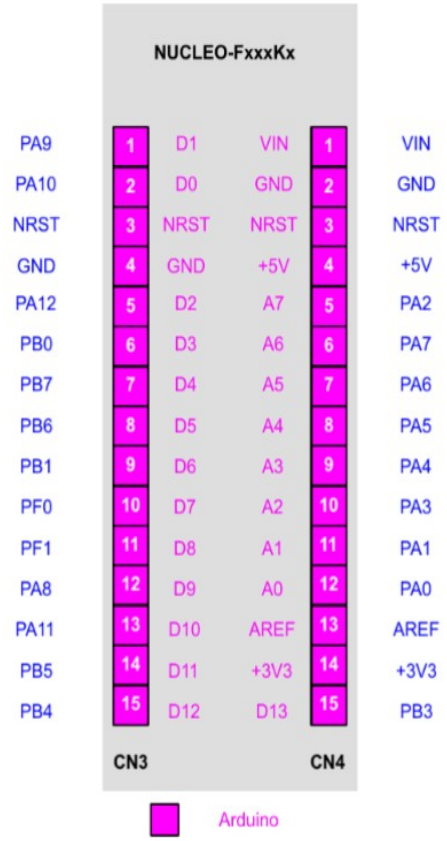
The pattern of 1's and 0's is transferred to memory location 0x50000414 as before but now something extra happens: signals (highs and lows) appear on pins on the side of the MCU. One of these signals is bit 3 which happens to be wired to LED LD3 on the Nucleo board. The value of b3 is 1 so the signal outside the chip is high which will turn on the LED. Writing a different number with a 0 in b3 will turn off LD3.

An LED is only one example of the kinds of things that can be controlled by ports. With amplification (signal strengthening) larger objects such as motors and heaters can also be controlled.

Now back to that word 'volatile'. C-compilers are very good at optimizing code so that it runs as quickly as possible (or takes up as little space as possible – there's a trade-off). If we write a program that simply writes values to address 0x50000414 and never reads them back the compile may decide that these data writes are not necessary as they are never read back. It simply won't include these instructions in the final executable program. To prevent it optimizing out these writes we declare the pointer GPIOB_ODR as a **pointer to a volatile int**. The keyword turns off optimization for reads and writes to this memory location so they will remain in the program.

The STM32F031 can have up to 6 ports, A to E and port H. Each of these ports can have up to 16 bits. When we write to a port's output register we write in blocks of 16 bits – i.e. 16 signals are written in parallel. This is why they are called parallel ports (we will look at serial ports later). There are several versions of the STM32F031 each with different numbers of pins. The particular one we are using is shown below:

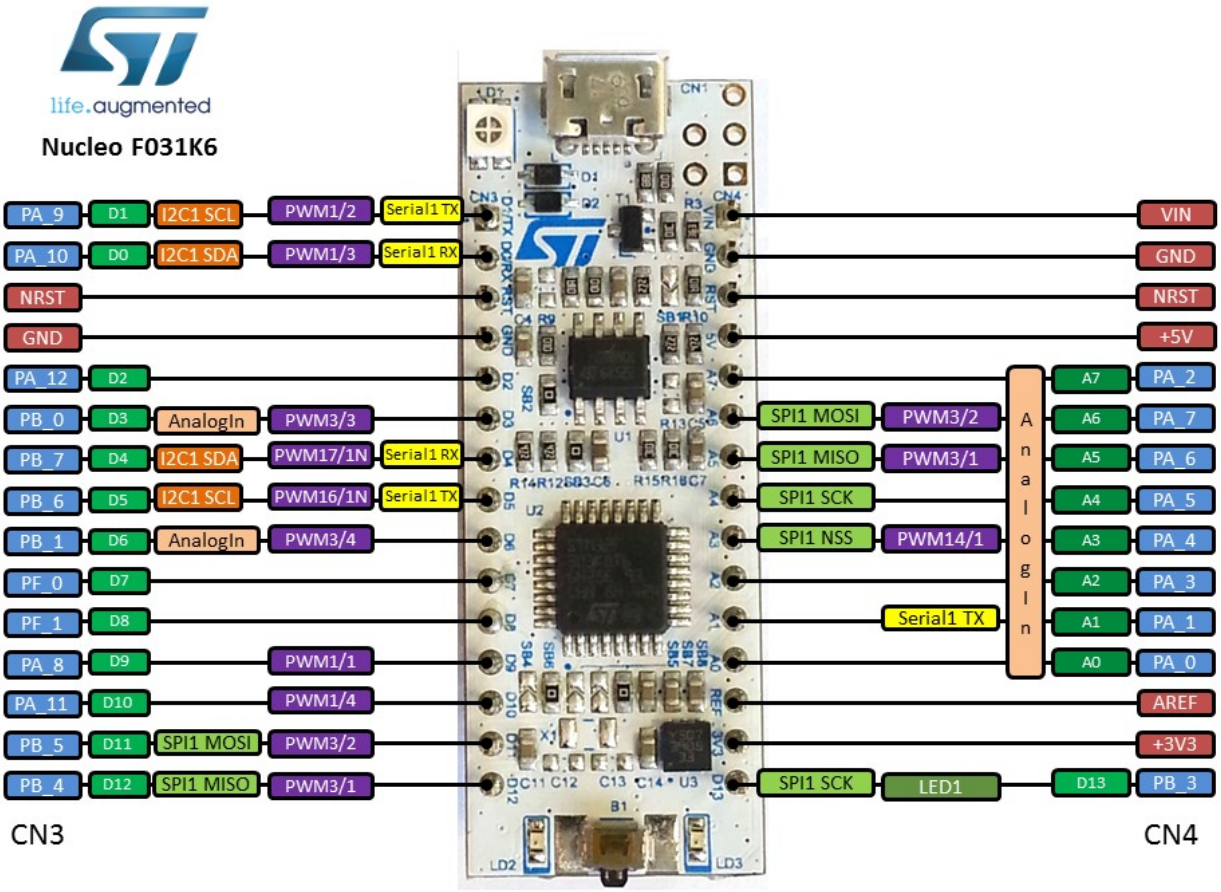
The pin labels we are interested in for now are the outermost ones (light blue). As you can see, only three ports are (partially) available: Ports A, B and C



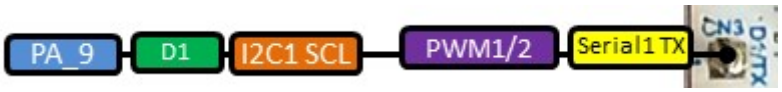
The manufacturer does not make all port pins available on every version of the device as it would cost too much as well as greatly complicate the circuit board design process. Engineers choose the particular version of the chip that best suits there needs.

Multi-function pins

The STM32F031-Nucleo board is often shown with this image:



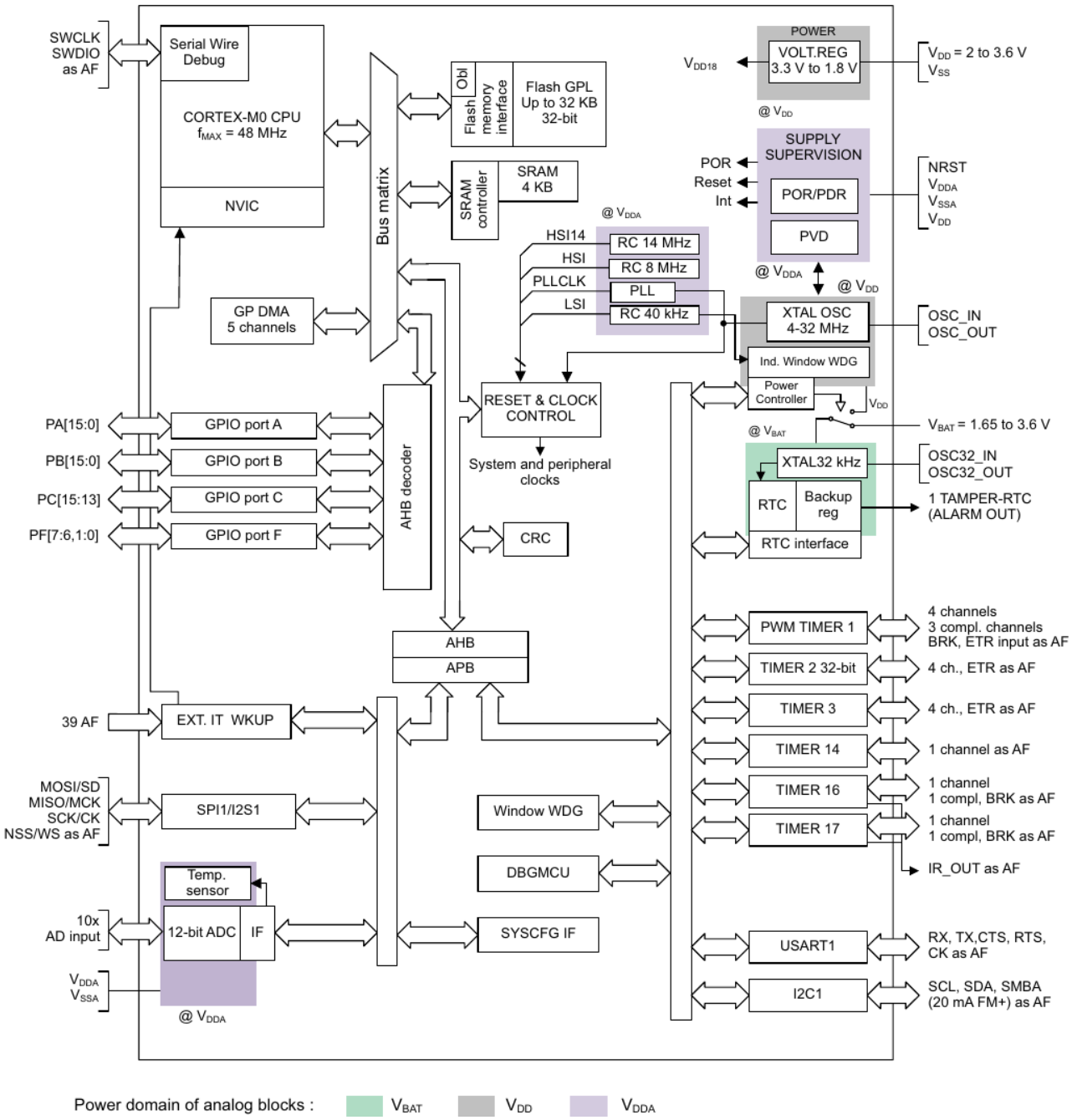
Let's look at one of these:



This pin is connected to bit 9 of Port A (PA_9) in the STM32F031. This pin also has other functions:

Transmit for UART 2(UART2_TX), PWM output number 1 from Timer 1 (PWM 1/2) and the serial clock output for I2C1 (I2C1_SCL). Don't worry what these particular subsystems are for now, we will meet some of them later. The important point to note is that pins can be reassigned to have different functions.

The figure above shows the contents of the STM32F031 MCU. There are lots of internal subsystems many of which can interact with pins on the side of the chip. It is cheaper for the manufacturer to design a single integrated circuit and package it in a number of different ways than to make custom chips for each end-user. The end-user then decides which components to enable and connect to the available pins. The unused subsystems simply remain turned off in the end-user designed product.



Pin modes

As we saw above, pins can be assigned to a range of different internal subsystems. They can also be defined as output or input pins. This is configuration is managed using **Mode** and **Alternative Function** registers – there is a set of these for each port.

Let's look at Port B for now.

The figure below is taken from page 158 of the reference manual

Suppose we would like to configure Port B, bit 3 (PB_3) as an output bit. There are two bits we need to configure MODE3[1:0].

If both of these bits are 0, then PB_3 will be a simple digital input

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 31:0 **MODER[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
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

If we have 01 in these bits then PB_3 will be a simple digital output (this is what we want for now).

So, we need to write the binary value:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 **01** 00 00 00 (this is 64 in decimal)

We can do this as follows:

```
volatile int *GPIOB_MODER = (volatile int *)0x48000400;
*GPIOB_MODER = 64;
```

Other pin modes are available to us but for now we will work with simple digital inputs and outputs only.

Turning on Parallel I/O ports.

After power-up, most of the internal sub-systems of our MCU are turned off (this saves power). We must turn them on before we configure and use them. In the case of the parallel input/output ports, there is a special register in the clock control system called IOPENR (Input/Output Port ENable Regiser). It is described in the reference manual (Page 193) as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TSCEN	Res.	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.
							rw		rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRC EN	Res.	FLITF EN	Res.	SRAM EN	DMA2 EN	DMA EN
									rw		rw		rw	rw	rw

In order to turn on Port B, we need to write a ‘1’ to bit 1 of this register. This is done as follows:

```
volatile int *RCC_AHBENR= (volatile int *)0x40021014;
*RCC_IOPENR = 2;
```

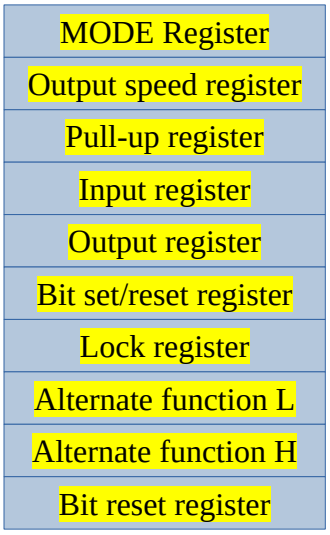
A complete program to blink LD3 at a human speed is shown below. Copy this in to your IDE and run on the Nucleo board. Vary the length of the delay and watch what happens. Question: Why is the value 8 written to the port output register?

```
void delay(volatile int dly);
int main()
{
     // RCC is in the region 0x40021000
     volatile int *RCC_IOPENR= (volatile int *)0x40021014;
     // GPIOB is in the region 0x50000400
     volatile int *GPIOB_MODER = (volatile int *)0x48000400;
     volatile int *GPIOB_ODR = (volatile int *)0x48000414;
     *RCC_IOPENR = 2;
     *GPIOB_MODER = 64;

     while(1)
     {
          *GPIOB_ODR = 8;
          delay(100000);
          *GPIOB_ODR = 0;
          delay(100000);
     }
}
void delay(volatile int dly)
{
     while(dly--);
}
```

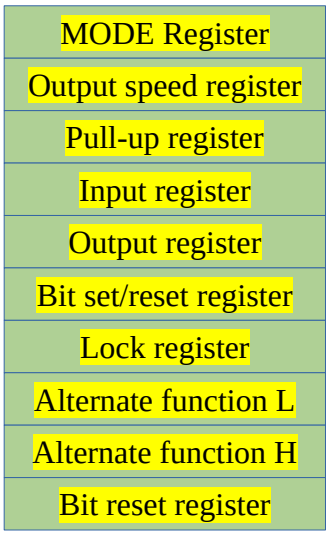
Working with Input/Output using structures

The example above does indeed manage to blink the LED onboard the nucleo board. It does however require you to go through the datasheets looking for addresses of the various subsystems within the MCU. This is cumbersome and error prone. STMicroelectronics, like a lot of manufacturers has released a number of C/C++ header files to help simplify the process. These header files make use of structures to map out the input/output address space. Lets take a look at the general purpose (parallel) input/output ports.

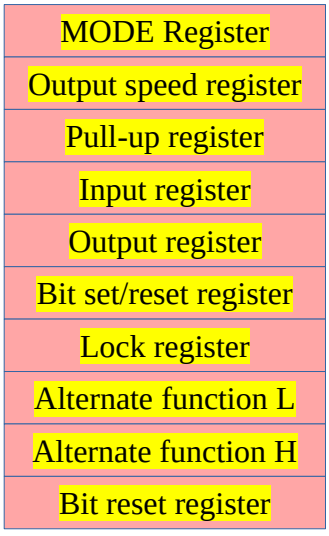
Port A: Starts at 0x48000000



Port A: Starts at 0x48000400



Port C: Starts at 0x48000800



As you can see, the layout of the registers associated with each port is identical. This means that the same data structure 'type' can be used for each one by using pointers and simply setting the pointer address to the start of the relevant register block. The header file for our device is called **stm32f031x6.h**. The section that deals with general purpose input/output is as follows:

```
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/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 AFRL;  /*!< GPIO alternate function register, Address offset: 0x20*/
  __IO uint32_t AFRL;  /*!< GPIO alternate function register, Address offset: 0x24 */
  __IO uint32_t BRR;     /*!< GPIO bit reset register, Address offset: 0x28 */
}GPIO_TypeDef;
```

The prefix "_IO" is the same as the keyword 'volatile'. Later on in this header files, pointers to structures of this type are declared and made to point at the corrector addresses for ports A,B, C etc. as follows:

```
#define GPIOA  ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB  ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC  ((GPIO_TypeDef *) GPIOC_BASE)
```

Other sub-systems within our MCU are declared in the same way. This allows us to rewrite the LED blink from above as follows:

```
#include <stdint.h>
#include <stm32l031xx.h>
void delay(volatile int dly);
int main()
{
     RCC->IOPENR = 2; // Turn on GPIOB
     GPIOB->MODER = 64; // Make GPIOB_3 an output

     while(1)
     {
          GPIOB->ODR = 8; // Make GPIOB_3 high
          delay(100000);  // wait a while
          GPIOB->ODR = 0; // Make GPIOB_3 low
          delay(100000);  // wait a while
     }
}
void delay(volatile int dly)
{
     while(dly--);
}
```

This is a lot simpler, readable and less error prone. This is the approach we will be using for the rest of this module.

Simplifying Port configuration.

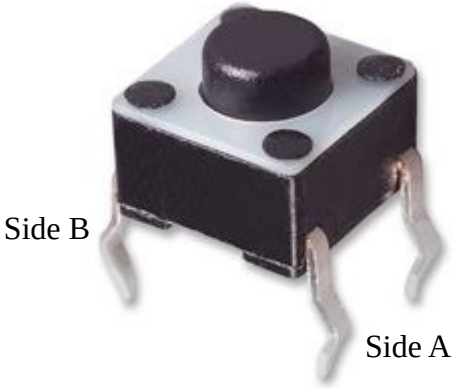
The above description may a lot to take in at first – when you get used to programming this device it should become more natural however, the function **pinMode** shown below should help to make life a little easier. This example is a further refinement of the blink program above.

```c
#include <stdint.h>
#include <stm32f031x6.h>
void pinMode(GPIO_TypeDef *Port, uint32_t BitNumber, uint32_t Mode);
void delay(volatile uint32_t dly);

int main()
{
	RCC->IOPENR = 2;   // Turn on GPIOB
	pinMode(GPIOB, 3, 1); // Make GPIOB_3 an output
	while(1)
	{
		GPIOB->ODR = 8; // Make Bit 3 high (8 = 2^3)
		delay(100000);  // Wait
		GPIOB->ODR = 0; // Make Bit 3 Low
		delay(100000);  // Wait
	}
}

void pinMode(GPIO_TypeDef *Port, uint32_t BitNumber, uint32_t Mode)
{
	// This function writes the given Mode bits to the appropriate location for
	// the given BitNumber in the Port specified.  It leaves other bits unchanged
	// Mode values:
	// 0 : digital input
	// 1 : digital output
	// 2 : Alternative function
	// 3 : Analog input
	uint32_t mode_value = Port->MODER; // read current value of Mode register
	Mode = Mode << (2 * BitNumber);    // There are two Mode bits per port bit
	                                   //so need to shift

	// the mask for Mode up to the proper location
	mode_value = mode_value & ~(3u << (BitNumber * 2)); // Clear out old mode bits
	mode_value = mode_value | Mode; // set new bits
	Port->MODER = mode_value; // write back to port mode register
}
void delay(volatile uint32_t dly)
{
	while(dly--);
}
```

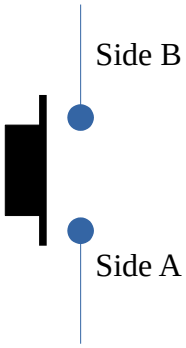
Digital Inputs

We have seen how general purpose output pins can be used to control LED's (and potentially other devices). We will now look at digital inputs – specifically buttons to see how our MCU handles them

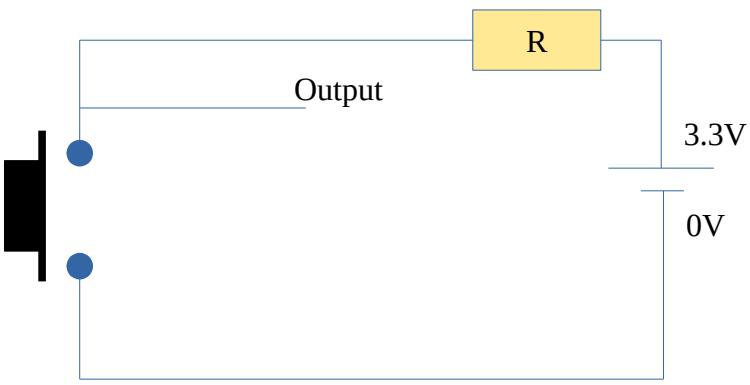
Turning a mechanical movement in to an electrical signal.



The push button shown above is a "momentary" push-button i.e. when pushed it connects the pins on side A to side B; once released the connection is broken. Electrically we can represent this as follows:



We can use this movement to create an electrical signal by connecting the button like this:

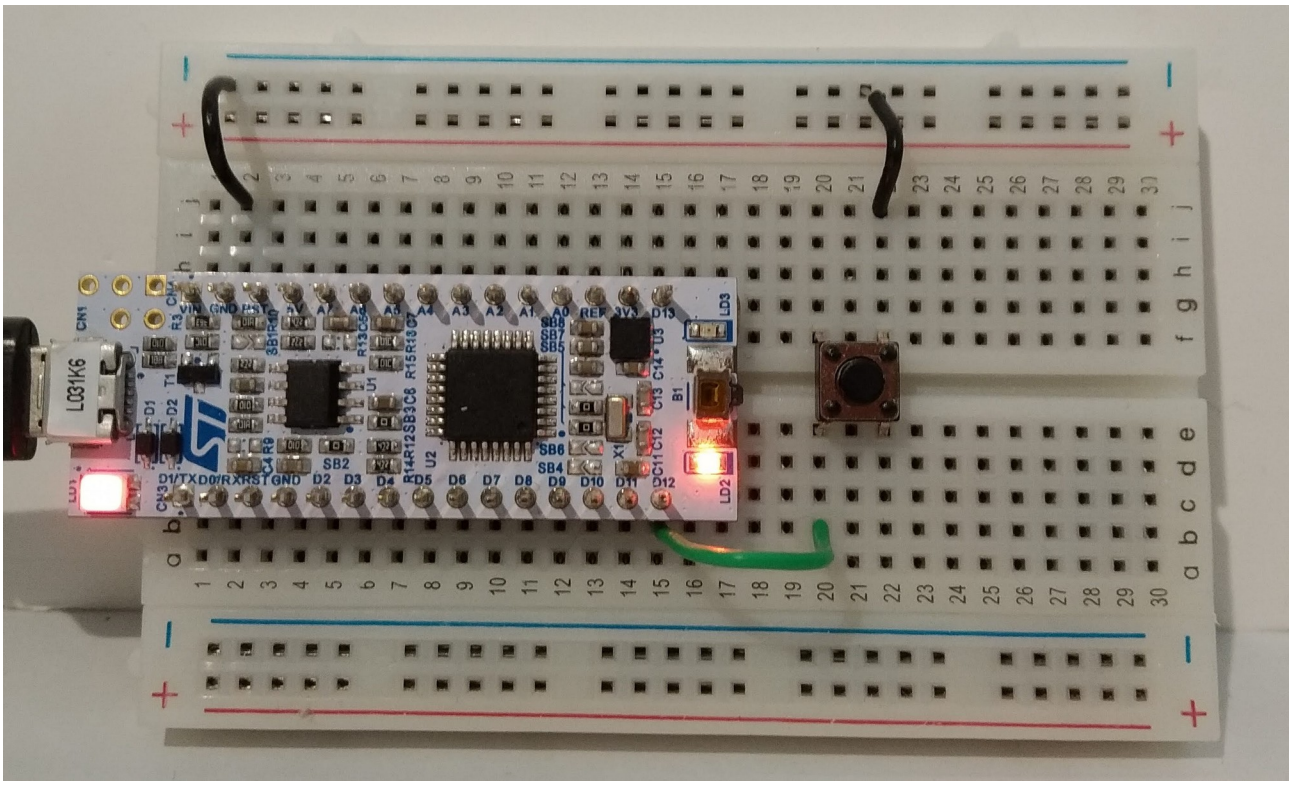


When the button is pressed, the Output signal wire is connected to 0V i.e. the output is Low. When the button is NOT pressed, the Output is connected via a Resistor (R) to 3.3V – the output is now High. The resistor is present to keep current flow to a low level when the button is pressed.

Resistors used in this fashion with buttons are often referred to as **Pull-Up** resistors. This is such a widely used meme that MCU manufacturers now often include the Pull-Up resistors in their devices. These internal Pull-Up resistors are usually disabled by default to save power but can be enabled by setting particular register bits. The STM32F031's internal pull-up resistors can be configured to pull a signal high or low depending on the setting of a pair of bits for each input. In the example that follows, a helper function called enablePullUp is included to simplify pull-up resistor configuration. Further details can be found in the reference manual page 224.

In this example you may notice that there is a 'u' after some of the constant values. This is to eliminate compiler warnings regarding signed/unsigned integer conversion. We will investigate this more later.

Example : Using a button to control the LED.



Code that turns on built-in LED when a button is pressed:

```
#include <stdint.h>
#include <stm32l031xx.h>
void pinMode(GPIO_TypeDef *Port, uint32_t BitNumber, uint32_t Mode);
void enablePullUp(GPIO_TypeDef *Port, uint32_t BitNumber);
void delay(volatile uint32_t dly);
int main()
{
	RCC->IOPENR = RCC->IOPENR | (1 << 1); // Enable GPIOB
	pinMode(GPIOB,3,1); // Make GPIOB_3 (LD3) an output
	pinMode(GPIOB,4,0); // Make GPIOB_4 (Button) an input
	enablePullUp(GPIOB,4); // enable pull-up for GPIOB_4
	while(1)
	{
		if (0 == (GPIOB->IDR & (1u << 4)))
		{
			// Button pressed
			GPIOB->ODR = GPIOB->ODR | (1u << 3); // LD3 On
		}
		else
		{
			// Button not pressed
			GPIOB->ODR = GPIOB->ODR & ~(1u << 3); // LD3 Off
		}
	}
}
void enablePullUp(GPIO_TypeDef *Port, uint32_t BitNumber)
{
	Port->PUPDR = Port->PUPDR & ~(3u << BitNumber*2); // clear pull-up resistor bits
	Port->PUPDR = Port->PUPDR | (1u << BitNumber*2); // set pull-up bit
}
void pinMode(GPIO_TypeDef *Port, uint32_t BitNumber, uint32_t Mode)
{
	// This function writes the given Mode bits to the appropriate location for
	// the given BitNumber in the Port specified.  It leaves other bits unchanged
	// Mode values:
	// 0 : digital input
	// 1 : digital output
	// 2 : Alternative function
	// 3 : Analog input
	uint32_t mode_value = Port->MODER; // read current value of Mode register
	Mode = Mode << (2 * BitNumber);    // There are two Mode bits per port bit so need to shift

	   // the mask for Mode up to the proper location
	mode_value = mode_value & ~(3u << (BitNumber * 2)); // Clear out old mode bits
	mode_value = mode_value | Mode; // set new bits
	Port->MODER = mode_value; // write back to port mode register
}
void delay(volatile uint32_t dly)
{
	while(dly--);
}
```

Input/Output Masks

What does the '=' sign do?

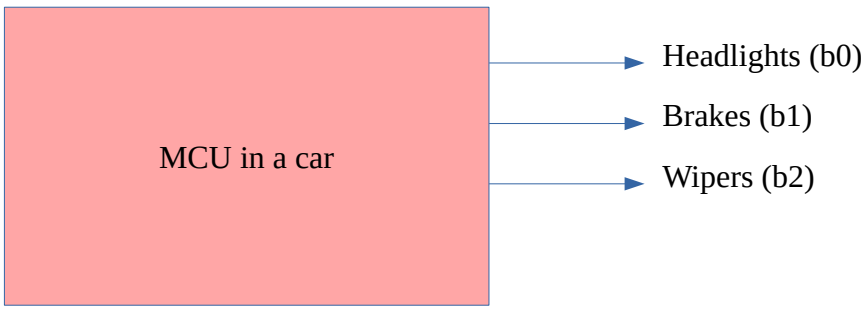
Consider the following short program:

```
#include <stdint.h>
uint8_t X;
uint16_t Y;
uint32_t Z;
int main()
{
    X = 42;
    Y = 42;
    Z = 42;
}
```

The values X, Y and Z are each assigned the value 42 so you might think that the same code underlies each of these operations and that the same amount of data is moved around in each case. This is not correct. When the compiler generates machine code that assigns 42 to X, it creates machine code that copies 8 bits of data into the memory location where X lives. In the case of Y, 16 bits of data are moved; Z : 32 bits. [2]

So, at the very least 8 bits of data in the destination variable are overwritten.

What if we want to write to just a single bit?



Imagine we had a microcontroller in a car that controls various devices as shown above. Lets suppose that all of these outputs are connected back to the same input/output port with bit numbering as shown. To turn on the headlights, the following code could work:

```
PORT = 1; // Set bit 0
```

To turn on the Brakes, you could do this:

```
PORT = 2; // Set bit 1
```

2 These sort of data movements are probably what actually happens in an 8 bit system. In a 32 bit system, the compiler may perform some optimization that leads to all writes being 32 bits

To turn on the wipers, you could do this:

```
PORT = 4; // Set bit 2
```

Now, you may have noticed that when you turn on the headlights, you also turned off the wipers and the brakes. Similarly, turning on the wipers turns off the brakes and headlights. This is not great. The C language has no built-in way of writing a single bit so how do we turn on the headlights for example without turning off the brakes?

The usual solution typically involves a *read-modify-write* process: Read the current value of the port into a variable, modify it in some way, and then write it back to the port. Lets take a close look at turning on the headlights.

The headlights turn on when we make Bit 0 of the port a 1.

Port output register

?	?	?	?	?	?	?	0

All of the bits marked with a '?' are unknown but should not be changed. How can we get a '1' in to bit 0 without affecting the other bits. One possible solution is to do an add.

```
X = Port;
X = X  + 1;
Port = X;
```

Port output register

?	?	?	?	?	?	?	0

\+

0	0	0	0	0	0	0	1

==========

?	?	?	?	?	?	?	1

In this particular case, this would work fine. There is a potential problem however. Supposing the headlights are already on and you repeat this operation again.

```
X = Port;
X = X  + 1;
Port = X;
```

Port output register

?	?	?	?	?	?	?	1

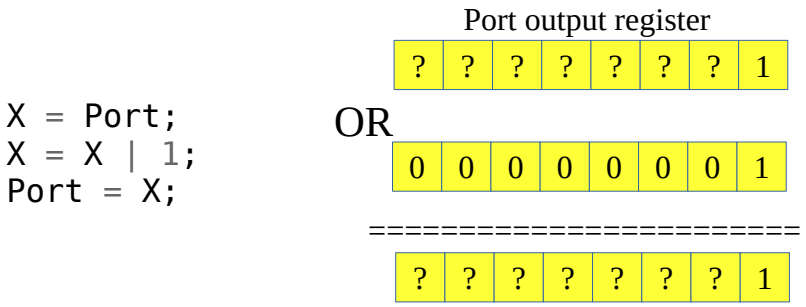
\+

0	0	0	0	0	0	0	1

==========

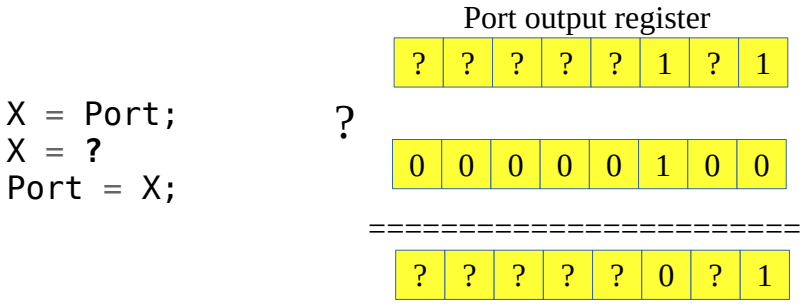
?	?	?	?	?	?	1	0

Instead of the headlights coming on, they go off and the brakes turn on! Simple addition will not work here – we need to use some of C's bit operations; specifically **OR**.

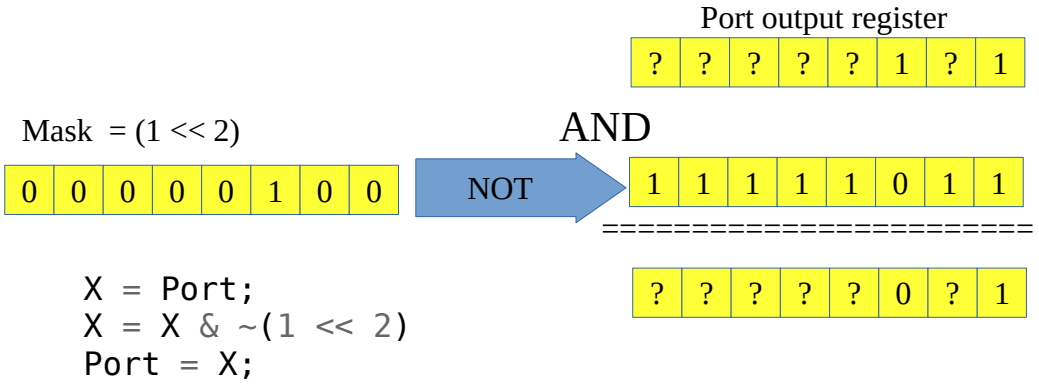


Using an OR like this will work whether Bit 0 is a 1 or 0 to start with. What would you OR X with to set Bit 5? Well 2^5 or 32 is the value but working out powers like that can be a little tricky and error prone. Happily the C language provides an easy way to do this: (1 << 5) i.e. Take the value of '1' and move it across the byte by 5 places (which makes it equivalent to 2^5). These values that we use to change specific bits are often called *bit-masks* or just *masks*.

Going back to our example suppose we want to turn off the wipers but leave the headlights on. What operation do we do here?



At first glance you may think that a subtract would be the thing to do. This would indeed work in this case however if the wipers were already off you would risk changing other bits (and turning the wipers back on again). So for the same reasons as additions didn't work before, subtracts won't work here either. In order to ensure that we change only the bit (or bits) in question we need to change the way we use the mask value and use an AND operation.



We want to change Bit 2 so we need a mask value of (1 << 2). Before we use this we need to invert all bits in this mask. This make Bit 2 of the mask a 0 and all other bits 1. When we AND this with the resultant Bit 2 will be zero and no other bits will be changed.

You can include multiple bits in a mask if you need to set several at the same time.