Digital Input and Output
Module Syllabus

- Digital Input and Output
  - Voltages and Logic Values

- GPIO Controller
  - Using Pointer to Access GPIO
  - Define Data Structure for Peripherals

- Digital IO Examples
  - Using LED
  - Using 7-Segment Display
  - Using Infrared Emitter/ Detector
Basic Programming – Digital IO
Voltages and Logic Values

- In digital devices, logic values (‘1’ or ‘0’) are represented as electrical voltages.
- Different devices may use different voltages to represent a logic value.
  - For example, the external pins of Nucleo F401RE platform use 3.3 volts for logic ‘1’, and 0 volts for logic ‘0’.
- Digital logic can however have different meanings in different contexts depending on the interpretation we give to different voltage levels.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Voltage</th>
<th>Boolean</th>
<th>Circuit</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘1’</td>
<td>3.3V</td>
<td>True</td>
<td>Closed</td>
<td>On</td>
</tr>
<tr>
<td>‘0’</td>
<td>0V</td>
<td>False</td>
<td>Open</td>
<td>Off</td>
</tr>
</tbody>
</table>
GPIO Design

- Normally, the external pins are not directly accessible, but accessed via a peripheral called General-purpose input/output (GPIO)
  - Used for general purpose, no special usage defined
  - Widely used for most applications
  - The direction of input/output is controlled by the direction register
  - A mask register is often used to mask out certain bits
**GPIO Design**

- GPIO pins split into groups, called ports
  - Each port has 32 pins [31:0]
  - Each of the ports are connected to the microcontroller via the peripheral bridge

- To reduce power consumption clocks are turned off for particular areas
  - Default setting – clock is disabled
  - To use a peripheral or a resource clock has to be enabled for each item individually

- Each pin is connected to a multiplexer
  - Allows each pin to perform several functions
  - Optimizes functionality small packages
  - Signal multiplexer and other pin options can be configured in the Pin Control Register (PCR)
For example, in the Nucleo F401RE MCU, there are eight GPIO peripherals named from PORTA-PORTH, and each of them has 10 registers, which include:

- **Port Mode Register (MODER)** - configure the I/O direction mode (input/output/alternate/analog)
- **Output Type Register (TYPER)** – configure the output type of the I/O port (push-pull/open-drain)
- **Output Speed Register (OSPEEDR)** – configure the I/O output speed (2/25/50/100 MHz)
- **Pull-Up/Pull-Down Register (PUPDR)** – configure the I/O pull-up or pull-down (no pull-up, pull-down/pull-up/pull-down)
- **Input Data Register (IDR)** – contain the input value of the corresponding I/O port
- **Output Data Register (ODR)** – can be read and written by software (ODR bits can be individually set and reset by writing to the BSRR)
- **Bit Set/Reset Register (BSRR)** – can be used for atomic bit set/reset
- **Configuration Lock Register (LCKR)** – used to lock the configuration of the port bits when a correct write sequence is applied to bit 16
- **Alternate Function Low Register (AFRL)** – configure alternate function I/Os
- **Alternate Function High Register (AFRH)** – configure alternate function I/Os
Using Pointer to Access GPIO

- Normally, you can use a pointer to directly access a GPIO peripheral in either C or assembly. For example in C, to light an LED on PB_10:

```c
#define RCC_AHB1ENR (*((volatile unsigned long *) 0x40023830))
#define GPIOB_MODER (*((volatile unsigned long *) 0x40020400))
#define GPIOB_PUPDR (*((volatile unsigned long *) 0x40020408))
#define GPIOB_OSPEEDR (*((volatile unsigned long *) 0x4002040c))
#define GPIOB_ODR (*((volatile unsigned long *) 0x40020414))

RCC_AHB1ENR |= 0x02;
GPIOB_MODER |= 0x00100000;
GPIOB_PUPDR |= 0x00200000;
GPIOB_OSPEEDR |= 0x00200000;
GPIOB_ODR |= 0x0400;
```

- This solution is fine for simple applications. However, if multiple instantiations of the same type of peripheral are available at the same time, we will need to define registers for each peripheral, which makes code maintenance difficult.

- On the other hand, since each register is defined as a separate pointer, each register access requires a 32-bit address constant. As a result, the program image will consume a larger memory space.
Define Data Structure for Peripherals

- To further simplify the code and reduce its length, we can:
  - Define the peripheral register set as a data structure,
  - Define the peripheral as a memory pointer to this data structure:

- For example:

```c
typedef struct {
    volatile unsigned int MODER;
    volatile unsigned int OTYPER;
    volatile unsigned int OSPEEDR;
    volatile unsigned int PUPDR;
    volatile unsigned int IDR;
    volatile unsigned int ODR;
    volatile unsigned int BSRRL;
    volatile unsigned int BSRRH;
    volatile unsigned int LCKR;
    volatile unsigned int AFR[2];
} GPIO_TypeDef;
```
Define Data Structure for Peripherals

- Then, to turn on an LED on PB_10:

```c
#define GPIOB_BASE 0x40020400
#define GPIOB((GPIO_TypeDef *) GPIOB_BASE)

RCC -> AHB1ENR |= 0x02;
GPIOB -> MODER |= 0x00100000;
GPIOB -> PUPDR |= 0x00200000;
GPIOB -> OSPEED |= 0x00200000;
GPIOB -> ODR |= 0x0400;
```
Define Data Structure for Peripherals

- With such arrangement:
  - The same register data structure for the peripheral can be shared between multiple instantiations
  - Hence code maintenance is easier
  - The requirement for immediate data storage is reduced
  - Compiled code is smaller, giving better code density

- With further modification, the functions developed for one peripheral can be shared between multiple instantiations by passing the base pointer to the function, for example:

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

void GPIO_init (GPIO_TypeDef *GPIO_pointer) {
  GPIOB -> MODER |= 0x00100000;
  GPIOA -> MODER |= 0x00010000;
}
```
Digital IO Example: LEDs

- Light-Emitting Diode (LED)
  - Emits light when switched on
  - Simplest way to indicate the status of logic
  - Also widely used in applications such as automotive lighting, general lighting, traffic signals etc.

Digital output sources current to LED

Digital output sinks current from LED
Digital IO Example: 7-Segment Display

- Use 7 segments and a dot to display numerals or letters
- Widely used in digital electronic devices, such as digital clocks, electronic meters
- Simple control, easy to debug
- Different values can be represented by different combinations of 8-bit segments (including dot), for example:

<table>
<thead>
<tr>
<th>Display</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit value</td>
<td>00111111</td>
<td>00000110</td>
<td>01011011</td>
<td>01001111</td>
<td>01100110</td>
</tr>
<tr>
<td>Display</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>8-bit value</td>
<td>01101101</td>
<td>01111101</td>
<td>00000111</td>
<td>01111111</td>
<td>01101111</td>
</tr>
</tbody>
</table>
Digital IO Example: Infrared Emitter/Detector

- **Infrared emitter (LED)**
  - A light-emitting diode that emits invisible infrared (IR) when conducting e.g. when connected to digital output:
  - ‘1’ – IR emitted (or brightness above a threshold)
  - ‘0’ – no IR emitted (or brightness below a threshold)

- **Infrared detector (photodiode)**
  - Converts light into either current or voltage e.g. when connected to digital input:
  - ‘1’ – IR received (or brightness above a threshold)
  - ‘0’ – no IR received (or brightness below a threshold)
Useful Resources

- Cortex-M4 Technical Reference Manual:
- Cortex-M4 Devices Generic User Guide:
- STM32 Nucleo Reference Manual: