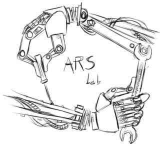
# The I$^2$C BUS Interface

Corrado Santoro

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

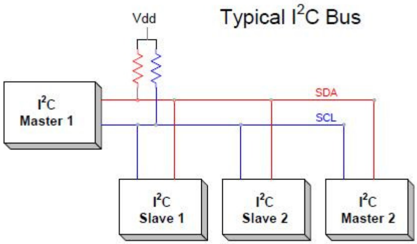


L.S.M. 1 Course

## What is I$^2$C?

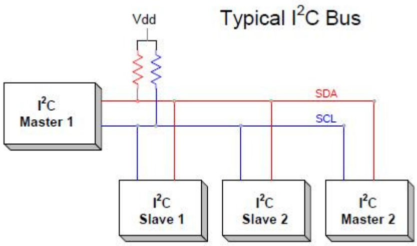
- **I$^2$C Bus** or **IIC Bus** is the acronym for **Inter-Integrated Circuit Bus**.
- It is a standard digital **communication bus** designed to interconnect *integrated circuits* belonging to the same board.
- It has been introduced by Philips to interconnect integrated circuits in TV-sets in the ’80s in the transition from discrete transistors to integrated-circuits.
- The bus has been initially used in TV-sets and VCRs, and then widely adopted in any integrated device which needs data communication.

## I$^2$C BUS: Philosophy and Connections



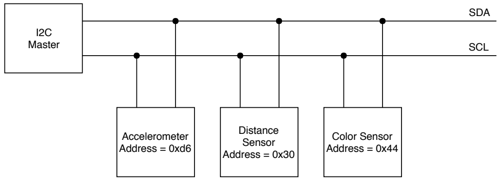
- I$^2$C has a **two wires bus** which interconnect all devices
- Devices in a I$^2$C network has a role:
  - **Master**, is the "head" of the bus and has the responsibility of starting a communication; only one master can be present in a I$^2$C network and is - in general - a MCU;
  - **Slave**, all the other devices which "respond" to master solicitations.

## I$^2$C BUS: Philosophy and Connections



- The I$^2$C wires have the following meaning:
  - **SDA: Serial DAta**, bidirectional; here data bits flow serially (one bit at time)
  - **SCL: Serial CLock**, undirectional from master to slaves; it holds the timing of the transmission
- Therefore I$^2$C is a **synchronous interface** which (according to standards) can reach the max speed of 400 *Kbps*

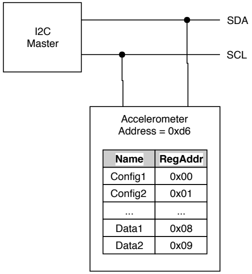
## I$^2$C Addressing



I2C Master
SDA
SCL
Accelerometer Address = 0xd6
Distance Sensor Address = 0x30
Color Sensor Address = 0x44

- Each **slave** device in I$^2$C has a well-know **address**
- The standard specifies two types of addresses:
  - **7-bit**, widely used
  - **10-bit**, used only in some special cases
- An additional bit (at B0) is added to the address to specify **direction** of transfer:
  - **“0”** = write to slave device
  - **“1”** = read from slave device
- Example: the accelerometer has address 0xd6:
  - **1 1 0 1 0 1 1 0** = 0xd6 = write to accelerometer
  - **1 1 0 1 0 1 1 1** = 0xd7 = read from accelerometer

# I$^2$C Addressing



| Name | RegAddr |
|---|---|
| Config1 | 0x00 |
| Config2 | 0x01 |
| ... | ... |
| Data1 | 0x08 |
| Data2 | 0x09 |

- Each **slave** device has also a **register map**
- Each register is identified by a 8-bit address and a 8-bit value
- Each register is used to:
  - Configure the device
  - Send commands to the device
  - Hold a sensed data
  - etc.
- Each register can be **read** or **written** from the master through proper transaction protocols.

## I$^2$C Transaction Model

A **Write Transaction** uses the following sequence:

1. Master sends the **START CONDITION (S)**
2. Master sends the **device address** (with B0 = 0)
3. Master sends the **register number** to be written
4. Master sends the **register data** to be written
5. Previous point is repeated for each subsequent register to be written
6. Master sends the **STOP CONDITION (P)**

- For each written byte, the Slave replies with an **ACKNOWLEDGE**, that is a **bit 0** sent over SDA line

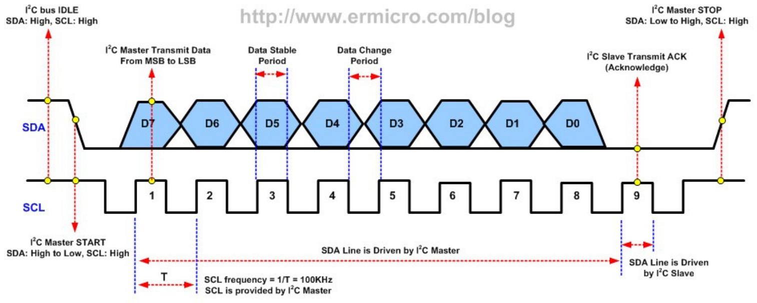
## I$^2$C Transaction Model

A **Read Transaction** uses the following sequence:

1. Master sends the **START CONDITION (S)**
2. Master sends the **device address** (with B0 = 0)
3. Master sends the **register number** to be read
4. Master sends a new **START CONDITION (S)**
5. Master sends the **device address** (with B0 = 1)
6. Salve sends the **register data** read
7. Previous point is repeated for each subsequent register to be read
8. Master sends the **STOP CONDITION (P)**

- For each byte transmitted over SDA, the peer device (Master or Slave) replies with an **ACKNOWLEDGE**, that is a **bit 0** sent over SDA line

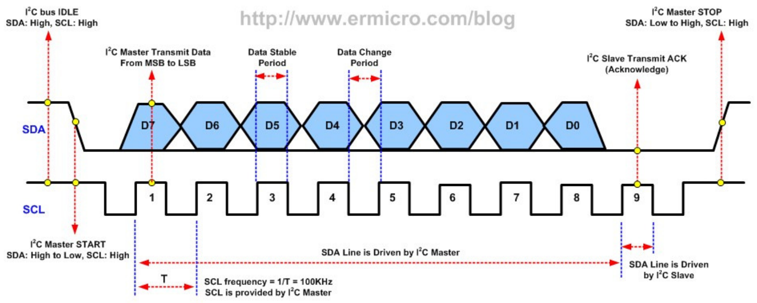
# I$^2$C: Timing and Bus States



I$^2$C Bus Master and Slave Timing Diagram

- **BUS IDLE**, both SDA and SCL lines are in 1 state.
- **START CONDITION (S)**
  - A transition **high-to-low** in SDA, while SCL is **high**, is a **Start Condition**
  - It is used to start communication on the bus
  - It is always initiated by the **Master**

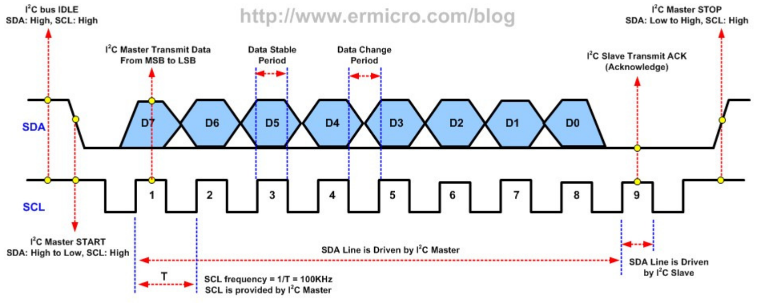
## $I^2C$: Data Transfer



$I^2C$ Bus Master and Slave Timing Diagram

- Data transfer occurs serially MSB-first:
  1. The bit value is set on the SDA line
  2. A pulse low-to-high-to-low occurs on the SCL line
  3. The next bit is sent ...
- After transmission of all the 8 bits, an **acknowledge** (ACK) is expected:
  1. The master generates a $9^{th}$ clock pulse
  2. The receiving device holds the SDA line **low** to signal that it has understood the byte sent

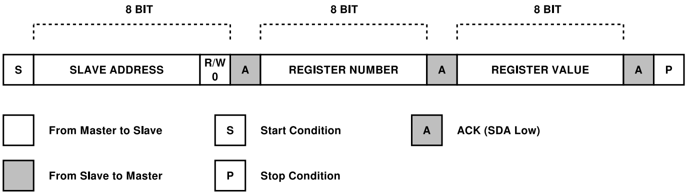
# I$^2$C: Stop Condition



I$^2$C Bus Master and Slave Timing Diagram

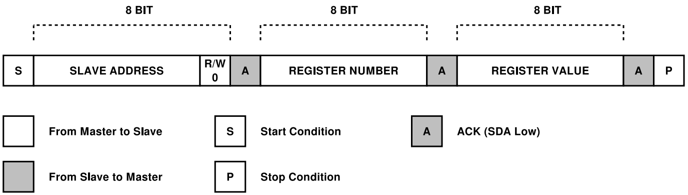
- When communication is over, a **STOP CONDITION (P)** is generated:
  - A transition **low-to-high** in SDA, while SCL is **high**, is a **stop condition**
  - It is used to stop any communication on the bus
  - It is always made by the **Master**
- After a Stop Condition, the bus goes in the Idle state.

## Sending Data to a Slave Device



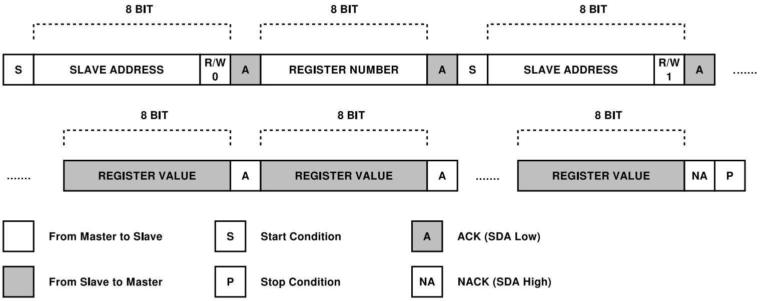
- First the Master initiates communication with a Start Condition
- The Master sends the **Write Command**, a 8-bit data, composed of:
  - The 7-bit address of the Slave device
  - The $R/\overline{W}$ bit at 0, which means **write-to-slave**
- The addressed Slave **acks**, by holding SDA line **low** in the $9^{th}$ clock pulse
- If no Slave exists at that address, the SDA line will remain to **high**, thus indicating a **NACK**; this situation is recognised by the Master which stops communication.

## Sending Data to a Slave Device



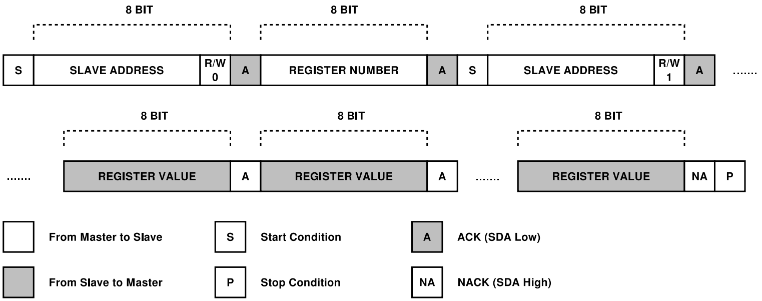
- After the address, the Master sends a 8-bit data which has the meaning of **register number**
- The addressed Slave **acks** data, by holding SDA line **low** in the 9th clock pulse
- Then the Master sends a 8-bit data which has the meaning of **register value**
- The addressed Slave **acks** data, by holding SDA line **low** in the 9th clock pulse
- The Master closes the transmission by sending a Stop Condition

## Receiving Data from a Slave Device



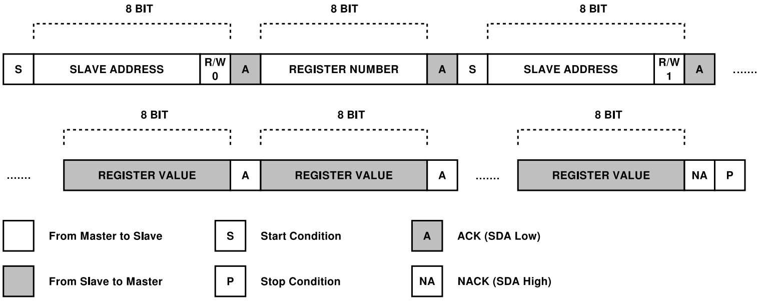
- First the Master initiates communication with a Start Condition
- The Master sends the **Write Command**, a 8-bit data, composed of:
  - The 7-bit address of the Slave device
  - The $R/\overline{W}$ bit at 0, which means **write-to-slave**
- The addressed Slave **acks**, by holding SDA line **low** in the 9$^{th}$ clock pulse

# Receiving Data from a Slave Device



- After the address, the Master sends a 8-bit data which has the meaning of **register number**
- The addressed Slave **acks** data, by holding SDA line **low** in the 9$^{th}$ clock pulse
- The Master sends a **new** Start Condition.
- The Master sends the **Read Command**, a 8-bit data, composed of:
  - The 7-bit address of the Slave device
  - The $R/\overline{W}$ bit at 1, which means **read-from-slave**
- The addressed Slave **acks**, by holding SDA line **low** in the 9$^{th}$ clock pulse

# Receiving Data from a Slave Device



- Slave device is now ready to send bytes
- The Slave sends a 8 bit data value
- The Master **acks**, by holding SDA line **low** in the 9$^{th}$ clock pulse
- The Slave sends the next 8 bit data value (next register value)
- The Master **acks**, by holding SDA line **low** in the 9$^{th}$ clock pulse
- When the Master is no more interested to data, it closes the communication by sending a **NACK** (holding SDA line **high** in the 9$^{th}$ clock pulse) and then a **Stop Condition**.

## The I2C functions of the stm32_unict_lib

- **Initialize the I2C interface:**
  `void I2C_init(I2C_TypeDef * port, int speed);`
  - **port**, the interface (usually `I2C1`)
  - **speed**, clock speed (e.g. 100000, 400000)

## The I2C functions of the stm32_unict_lib

- **Write a single register:**

```
void I2C_write_register(I2C_TypeDef * port,
                        short addr, short reg_adr,
                        short reg_val);
```

  - **port**, the interface (usually `I2C1`)
  - **addr**, device address
  - **reg_adr**, register address
  - **reg_val**, register value

- **Read a single register:**

```
void I2C_read_register(I2C_TypeDef * port,
                       short addr, short reg_adr,
                       short * reg_val);
```

  - **port**, the interface (usually `I2C1`)
  - **addr**, device address
  - **reg_adr**, register address
  - **reg_val**, pointer to the variable that will receive register value

## The I2C functions of the stm32_unict_lib

- **Write a set of registers:**

```
void I2C_write_buffer(I2C_TypeDef * port,
                      short addr, short reg_adr,
                      unsigned char * data, int len);
```

  - **port**, the interface (usually `I2C1`)
  - **addr**, device address
  - **reg_adr**, register address
  - **data**, the buffer holding register values
  - **len**, the buffer length (number of registers)

- **Read a set of registers:**

```
void I2C_read_buffer(I2C_TypeDef * port,
                     short addr, short reg_adr,
                     unsigned char * data, int len);
```

  - **port**, the interface (usually `I2C1`)
  - **addr**, device address
  - **reg_adr**, register address
  - **data**, the buffer that will receive register values
  - **len**, the buffer length (number of registers)

# The I$^2$C BUS Interface

Corrado Santoro

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

L.S.M. 1 Course