KYAMBOGO UNIVERSITY

Department of Electrical and Electronics Engineering

TETE 3102 MICROCONTROLLERS AND APPLICATIONS

Lecture Notes and Student Tasks: I2C Communication with STM32 and Arduino

Lecturer: Dr. Dickson Mugerwa

1. PURPOSE OF THESE NOTES

These notes expand the lecture slides "Serial Communication in STM32: I2C Communication (Inter-Integrated Communication)" into a full teaching text. They cover the protocol itself, then show how to use it on the STM32F411 (Nucleo-64 type board, STM32CubeMX and the HAL library) and on the Arduino Uno (Wire library). A teaching plan is in Section 10, student tasks are in Section 11 and short lecturer answers are in Section 12. Everything is written so that the lecturer can teach from it directly.

Note on the slides: several slide titles read "12C" (the digit one followed by the digit two). The correct name is I2C, pronounced "I squared C" or "I two C", short for Inter-Integrated Circuit. It was introduced by Philips (now NXP). The official specification is NXP document UM10204.

2. LEARNING OUTCOMES

By the end of the lecture and practical, a student should be able to:

- Explain what I2C is, name its two lines, and state its main properties (multi-master, multi-slave, synchronous, half-duplex).
- Explain why the lines are open-drain and need pull-up resistors, and calculate a suitable pull-up value.
- Convert between a 7-bit device address and the 8-bit address byte used on the wire, for both read and write.
- Draw and explain START, STOP, repeated START, ACK and NACK, and sketch a complete write and read transaction.
- Configure I2C1 in STM32CubeMX and communicate in polling, interrupt and DMA modes with the HAL library.
- Use the Arduino Wire library to scan the bus, read a register-based device and drive an I2C LCD.
- Find faults on a bus (wrong address, missing pull-ups, address conflicts) using a scanner, return codes and an oscilloscope or logic analyser.

3. WHY SERIAL BUSES, AND HOW I2C COMPARES

A microcontroller usually has to talk to many other chips: displays, sensors, real-time clocks, memories, converters. Connecting each one with a wide parallel bus would use up many pins.

Serial buses send the bits one after another over very few wires. The three buses students meet most often are UART, SPI and I2C.

Feature	UART	SPI	I2C
Wires	2 (TX, RX) per link	4 (SCLK, MOSI, MISO, CS); one CS per slave	2 (SDA, SCL) shared by all
Clock	Asynchronous (agreed baud rate)	Synchronous, master supplies	Synchronous, master supplies
Addressing	None (point to point)	Hardware chip select	7-bit (or 10-bit) address in the frame
Devices	Two	Many, but one CS pin each	Up to about 112 addresses, limited by 400 pF bus capacitance
Speed	Typically up to about 1 Mbit/s	Often 10 Mbit/s and more	100 kbit/s, 400 kbit/s, 1 Mbit/s (and faster modes)
Duplex	Full	Full	Half
Acknowledge	No	No	Yes, every byte
Best for	PC link, GPS, modules	Fast memory, displays, SD cards	Many slow sensors and small displays on few pins

I2C is chosen when pins are scarce and the data rate is modest. It is not the best choice when very high throughput is needed.

4. I2C FUNDAMENTALS

4.1 The two lines and the roles

- **SDA** (Serial Data): carries the data bits. Master and slave both drive it, one at a time, so it is bidirectional.
- **SCL** (Serial Clock): the clock that synchronises the data. Normally generated by the master.
- **Master**: starts and stops a transaction, generates the clock and addresses a slave.
- **Slave**: responds when its address is called. It may hold the clock low to ask the master to wait (clock stretching).

The bus is multi-master (more than one master may exist, with arbitration), multi-slave, synchronous, bidirectional and half-duplex: data goes in only one direction at any moment.

4.2 Open-drain outputs and pull-up resistors

No device ever drives SDA or SCL high. Each device has an open-drain (open-collector) output: it can pull the line to ground or release it. The pull-up resistors Rp, connected to VDD,

bring a released line to logic high. So a line is high when every device releases it, and low when any device pulls it down. This is a wired-AND connection.

This design is important for two reasons. First, two devices can never cause a short circuit by fighting (one driving high, the other low), which makes multi-master operation and clock stretching safe. Second, devices with different supply voltages can share a bus if the pull-ups go to a voltage they all tolerate. The slide states that the common pull-up value is 4.7 kohm and that the value depends on the bus capacitance and the speed. The lines idle high.

4.3 Choosing the pull-up value

Each line has capacitance Cb (wires, pins, connectors). The pull-up and Cb form an RC circuit, so the rising edge is slow. The I2C specification limits the rise time tr (from 30 % to 70 % of VDD): 1000 ns in Standard-mode, 300 ns in Fast-mode and 120 ns in Fast-mode Plus. The maximum bus capacitance is 400 pF. The rise time is:

$$t_r = 0.8473 \times R_p \times C_b$$

Where 0.8473 = ln(0.7/0.3). The upper limit of Rp therefore is:

$$R_{p(max)} = t_{r(max)} / (0.8473 \times C_b)$$

The lower limit comes from the current a device must sink while holding the line low. The specification asks a device to sink 3 mA at a low level of at most 0.4 V in Standard and Fast modes (20 mA in Fast-mode Plus):

$$R_{p(min)} = (V_{DD} - 0.4\ V) / 3\ mA$$

Worked example. VDD = 3.3 V and Cb = 100 pF.

- Rp(min) = (3.3 - 0.4) / 0.003 = 967 ohm.
- Rp(max) at 100 kHz = 1000 ns / (0.8473 x 100 pF) = 11.8 kohm.
- Rp(max) at 400 kHz = 300 ns / (0.8473 x 100 pF) = 3.54 kohm.

So 4.7 kohm is fine at 100 kHz but too large at 400 kHz for this bus (the rise time would be 0.8473 x 4700 x 100 pF = 398 ns, which is more than 300 ns). For 400 kHz a value between about 1 kohm and 3.3 kohm, for example 2.2 kohm, is appropriate. Practical rules: a long cable or many devices raises Cb and needs a smaller Rp; a small Rp wastes power and needs strong drivers. Many breakout modules already have pull-ups on board (often 4.7 kohm or 10 kohm). With several such modules on one bus these resistors are in parallel, so the effective value can become too low.

The internal pull-ups of microcontrollers (typically 20 kohm to 50 kohm and poorly defined) are too weak for reliable I2C except for very short wires at low speed. Use external resistors.

4.4 Addressing

Every slave has an address. A 7-bit address allows 128 values, but some are reserved: 0x00 to 0x07 and 0x78 to 0x7F are reserved (general call, CBUS, 10-bit addressing and so on), leaving

0x08 to 0x77, that is 112 usable addresses. The first byte after START carries the address in its upper seven bits (A6 to A0) and the direction in bit 0:

```
bit:    7  6  5  4  3  2  1  0
field:  A6 A5 A4 A3 A2 A1 A0 R/W
```

- R/W = 0: the master will write to the slave.
- R/W = 1: the master will read from the slave.

Only the slave whose address matches continues to take part, which avoids collisions. The formula for the byte on the wire is:

address byte = (7-bit address << 1) | R/W

Device (example)	**7-bit address**	**Write byte**	**Read byte**	**Comment**
Slide example, Slave 1 (0110000)	0x30	0x60	0x61	Slide
Slide example, Slave 2 (1001000)	0x48	0x90	0x91	Same as LM75, ADS1115, PCF8591 family
Slide example, Slave 3 (0101101)	0x2D	0x5A	0x5B	Slide
LCD1602 with PCF8574 backpack	0x27 (or 0x3F)	0x4E (0x7E)	0x4F (0x7F)	0x27 for PCF8574, 0x3F for PCF8574A in the default jumper setting
SSD1306 OLED	0x3C (or 0x3D)	0x78 (0x7A)	0x79 (0x7B)	The “0x78” on the board in the slide is the 8-bit form
DS3231 real-time clock	0x68	0xD0	0xD1	Fixed address

Important confusion to explain to students. Datasheets and board labels sometimes quote the 7-bit address (0x27, 0x3C) and sometimes the 8-bit write address (0x4E, 0x78). Arduino Wire uses the 7-bit form. The STM32 HAL expects the 7-bit address shifted left by one bit (so 0x27 is passed as 0x27 << 1 = 0x4E). Forgetting this shift is the most common reason an STM32 I2C program fails.

Many chips have two or three address pins (A0, A1, A2) that select the lower address bits, so several identical chips can share a bus. PCF8574, for example, uses 0x20 to 0x27 depending on A2 A1 A0. Two devices with the same fixed address cannot share one bus unless an I2C multiplexer (for example TCA9548A) is used.

4.5 Bus conditions: START, STOP and repeated START

- **START (S)**: SDA falls from high to low while SCL is high. This is the only time SDA is allowed to change while SCL is high (apart from STOP). It announces the beginning of a transaction and the bus becomes busy.
- **STOP (P)**: SDA rises from low to high while SCL is high. The bus becomes free.
- **Repeated START (Sr)**: a START sent without a STOP in between. It lets the master change direction (for example write a register address, then read the data) without releasing the bus to another master.

During data transfer the rule is: SDA may change only while SCL is low and must be stable while SCL is high. A bit is sampled on the rising edge of SCL.

4.6 Acknowledge and not acknowledge

Every byte is sent most significant bit first and is followed by a ninth clock pulse used for the acknowledge. The transmitter releases SDA during that ninth pulse and the receiver may pull it low:

- **ACK (A)**: SDA is low during the ninth clock. It means the byte was received.
- **NACK (/A, NAK)**: SDA stays high during the ninth clock. It means no device answered, the device is busy or cannot accept more data, or (in a read) that the master has received the last byte it wants.

A NACK after the address byte is how a scanner discovers that no device exists at that address. In a read transaction, the master ACKs every byte except the last, which it NACKs; it then sends STOP. That tells the slave to release SDA.

4.7 Complete transactions

Write transaction (master to slave), as on the slide:

S | A6..A0 | 0 | ACK | D7..D0 | ACK | D7..D0 | ACK | ... | P

The address byte has R/W = 0. The slave ACKs the address and then ACKs each data byte. The master finishes with STOP.

Read transaction (slave to master):

S | A6..A0 | 1 | ACK | D7..D0 | ACK | D7..D0 | NACK | P

The address byte has R/W = 1. After the slave ACKs, it sends data. The master ACKs each byte except the last, for which it sends NACK, then STOP.

Register read (combined transaction). Most sensors keep their data in registers. To read a register the master first writes the register number, then restarts and reads:

S | addr+W | ACK | reg | ACK | Sr | addr+R | ACK | data | NACK | P

This is exactly what HAL_I2C_Mem_Read does on the STM32 and what Wire.endTransmission(false) followed by Wire.requestFrom does on the Arduino.

Timing example. At 100 kHz one clock is 10 microseconds, and a byte plus its ACK takes 9 clocks = 90 microseconds. Writing two data bytes to a device takes address + 2 data = 3 bytes = 27 clocks = 270 microseconds, plus START and STOP. At 400 kHz the same takes about 67.5 microseconds per byte. The real throughput is lower than the headline bit rate because of the address byte and the ACK bit.

4.8 Speed modes

Mode	Maximum rate	Typical use
Standard-mode (Sm)	100 kbit/s	Default; long or lightly shielded wires; RTC, simple sensors
Fast-mode (Fm)	400 kbit/s	Most displays and sensors
Fast-mode Plus (Fm+)	1 Mbit/s	Needs stronger drivers (20 mA sink) and smaller Rp
High-speed (Hs) and Ultra-fast (UFm)	3.4 Mbit/s and 5 Mbit/s	Specialised; not covered in this course

All devices on the bus must support the chosen speed. One slow device limits the whole bus.

4.9 Clock stretching and multi-master arbitration

Clock stretching. A slave that needs more time can hold SCL low after the master releases it. The master must wait until SCL actually rises before continuing. The slide frame shows the clock line held low while the slave services an interrupt after a byte is complete. Some cheap microcontroller masters and software I2C routines ignore stretching and fail with such slaves.

Arbitration. If two masters start at the same time, both send their bits. Because the bus is wired-AND, the master that sends a 1 while seeing a 0 on SDA knows it lost, stops transmitting and waits. No data is lost. In practice most student projects use one master.

5. DEVICES USED IN THE LABORATORY

5.1 Display modules (from the slides)

- **Segment displays**: 7, 14 or 16 LEDs arranged in the shape of an 8, each segment controlled separately; common cathode and common anode types exist. Used on fuel price boards, clocks and meters. Multi-digit modules often have an on-board driver (for example TM1637 or MAX7219) with their own serial interface; some are I2C.
- **Character LCDs** (for example 16x2): show alphanumeric characters, many characters depending on size. Two types: parallel and serial.
- **Serial versus parallel LCDs**: a parallel LCD receives 4 or 8 data bits at a time and needs many pins (RS, E, D4-D7 at least six, plus power and contrast). A serial LCD receives the bits one at a time over clock pulses. A "1602 IIC" backpack achieves this: a PCF8574 I2C port expander on the back of the LCD converts two I2C wires into the parallel lines.
    - **Graphic OLED** (for example 0.96 inch, 128x64, SSD1306): self-lit pixels, I2C or SPI versions. The I2C version has four pins: GND, VCC, SCL, SDA.

Real-world uses shown on the slides include calculators, phone screens and roadside billboards.

5.2 LCD1602 with PCF8574 backpack

The HD44780-compatible LCD is controlled in 4-bit mode through the PCF8574, an 8-bit I/O expander. Writing one byte to the PCF8574 sets its eight output pins. The usual wiring of the backpack is:

PCF8574 pin	**P0**	**P1**	**P2**	**P3**	**P4 to P7**
LCD signal	RS	RW (held 0)	E (enable)	Backlight transistor	D4 to D7
Bit mask	0x01	0x02	0x04	0x08	0x10 to 0x80

To send a byte to the LCD in 4-bit mode: send the upper nibble with E = 1, then with E = 0 (the falling edge of E latches the data); then the lower nibble in the same way. RS = 0 selects a command, RS = 1 selects character data. The trimmer on the backpack adjusts contrast; if the display is lit but shows blocks or nothing, turn it. Most backpacks are designed for 5 V.

5.3 DS3231 real-time clock

The DS3231 (address 0x68) keeps time in registers coded in BCD (binary-coded decimal). Registers: 0x00 seconds, 0x01 minutes, 0x02 hours, 0x03 day of week, 0x04 date, 0x05 month, 0x06 year. Example: 45 seconds is stored as 0x45 (not 0x2D). Convert with dec = (v >> 4) * 10 + (v & 0x0F) and bcd = ((dec / 10) << 4) | (dec % 10). It is a good example of a register-based I2C device and is used in the tasks.

5.4 SSD1306 OLED

The common 128x64 module has I2C address 0x3C (printed as 0x78 on many boards, which is the 8-bit write form). Commands and pixel data are sent as control byte plus data. Students are expected to use a library (Adafruit SSD1306 and Adafruit GFX on Arduino; a ported SSD1306 driver on STM32) rather than write the driver.

6. I2C ON THE STM32 (STM32F411, CubeMX AND HAL)

6.1 Hardware

The STM32F411RE (LQFP64 package, as on the slide and on the Nucleo-F411RE board) has three I2C peripherals: I2C1, I2C2 and I2C3. The reference manual is RM0383 (the slide cites page 212, the I2C chapter). In the slide, I2C1 is enabled in CubeMX with the pins highlighted next to BOOT0 on the top edge of the chip. These are PB8 (I2C1_SCL) and PB9 (I2C1_SDA), alternate function AF4. On the Nucleo-64 board they appear on the Arduino-style header as D15 (SCL) and D14 (SDA). Check the pin names that CubeMX actually assigns in your project, because I2C1 can also be moved to PB6 (SCL) and PB7 (SDA).

The STM32 runs at 3.3 V. PB8 and PB9 are five-volt tolerant pins, but confirm this in the datasheet pin table (look for "FT") before connecting a 5 V module with its own pull-ups to 5 V. When in doubt, power modules from 3.3 V or use a bidirectional level shifter.

Connection	Notes
GND of module to GND of board	Always common ground
VCC of module to 3.3 V (or 5 V if designed for it)	LCD backpacks usually need 5 V for readable contrast
SDA to PB9, SCL to PB8	External pull-ups 4.7 kohm to 3.3 V if the module has none

6.2 Configuring I2C1 in STM32CubeMX

1. Create a new project and select the board (NUCLEO-F411RE) or the MCU STM32F411RETx.
2. Pinout and Configuration, Categories, Connectivity, I2C1. In Mode set I2C to "I2C" (this is the circled setting on the slide). CubeMX turns the two pins green.
3. Parameter Settings: I2C Speed Mode = Standard Mode, I2C Clock Speed = 100000 Hz (use Fast Mode 400000 Hz later). Leave 7-bit addressing. For Fast Mode keep the duty cycle at 2 unless you have a reason to change it.
4. GPIO Settings: check that SCL and SDA are alternate function, open drain, and set to pull-up or no pull. External pull-ups are better than the internal ones.
5. NVIC Settings (needed only for interrupt mode): tick I2C1 event interrupt and I2C1 error interrupt.
6. DMA Settings (needed only for DMA mode): Add I2C1_TX and I2C1_RX, direction memory to peripheral and peripheral to memory, mode Normal.
7. Optional for printing results: Connectivity, USART2, Mode Asynchronous, 115200 baud. This is the virtual COM port of the Nucleo ST-LINK; open it with PuTTY or the Arduino serial monitor.
8. Clock Configuration: the APB1 clock must be at least 2 MHz for Standard mode and 4 MHz for Fast mode. The default is far above this.
9. Project Manager: name the project, choose the toolchain (STM32CubeIDE), generate code. CubeMX produces MX_I2C1_Init() and the handle I2C_HandleTypeDef hi2c1 in main.c. Write your own code only between the "USER CODE BEGIN" and "USER CODE END" markers, otherwise regeneration deletes it.

6.3 The main HAL functions

Function	Use
HAL_I2C_Master_Transmit(&hi2c1, addr, buf, len, timeout)	Master writes len bytes to a slave (blocking)
HAL_I2C_Master_Receive(&hi2c1, addr, buf, len, timeout)	Master reads len bytes from a slave (blocking)

Function	Use
HAL_I2C_Mem_Write(&hi2c1, addr, reg, regsize, buf, len, timeout)	Writes a register address and then data in one transaction
HAL_I2C_Mem_Read(&hi2c1, addr, reg, regsize, buf, len, timeout)	Writes the register address, repeated START, then reads
HAL_I2C_IsDeviceReady(&hi2c1, addr, trials, timeout)	Sends the address only and checks for ACK; used by scanners
..._IT versions	Non-blocking; completion reported by a callback from the interrupt
..._DMA versions	DMA moves the bytes; the CPU is free; completion by callback
HAL_I2C_GetError(&hi2c1)	Returns error flags such as HAL_I2C_ERROR_AF (acknowledge failure), BERR, ARLO, OVR, TIMEOUT

Remember: addr is the 7-bit address shifted left by one bit, for example (0x68 << 1). The HAL sets the R/W bit itself. The return type is HAL_StatusTypeDef: HAL_OK, HAL_ERROR, HAL_BUSY or HAL_TIMEOUT. The timeout is in milliseconds.

The three driver styles named on the slide are:

- **Polling (blocking) mode**: the function does not return until the transfer ends or times out. Simple, but the CPU waits.
- **Interrupt mode**: the function starts the transfer and returns at once. The I2C interrupt handler moves the bytes and a callback tells the program when it is finished.
- **DMA mode**: like interrupt mode, but a DMA channel moves the data, so there is almost no CPU load per byte. Best for long transfers (for example refreshing a graphic OLED).

6.4 Example 1: I2C bus scanner (polling)

Add near the top of main.c in USER CODE BEGIN Includes: #include <stdio.h> and #include <string.h>. Then in main(), after the initialisation calls:

```
/* USER CODE BEGIN 2 */
char msg[64];
int n = snprintf(msg, sizeof msg, "I2C scan start\r\n");
HAL_UART_Transmit(&huart2, (uint8_t*)msg, n, 100);
for (uint8_t a = 8; a < 120; a++) {
   if (HAL_I2C_IsDeviceReady(&hi2c1, (uint16_t)(a << 1), 2, 10) == HAL_OK) {
      n = snprintf(msg, sizeof msg, "Found device at 0x%02X\r\n", a);
      HAL_UART_Transmit(&huart2, (uint8_t*)msg, n, 100);
   }
}
n = snprintf(msg, sizeof msg, "Scan done\r\n");
HAL_UART_Transmit(&huart2, (uint8_t*)msg, n, 100);
/* USER CODE END 2 */
```

How it works: IsDeviceReady sends START and the address byte only. If a device ACKs, the function returns HAL_OK. Expected output with an LCD backpack and an RTC on the bus: "Found device at 0x27" and "Found device at 0x68". Scan from 8 to 119 (0x08 to 0x77) to avoid reserved addresses.

6.5 Example 2: reading and setting the DS3231 clock with Mem_Read and Mem_Write

```
#define DS3231_ADDR  (0x68 << 1)

static uint8_t bcd2dec(uint8_t v) { return (uint8_t)((v >> 4) * 10 + (v & 0x0F)); }
static uint8_t dec2bcd(uint8_t v) { return (uint8_t)(((v / 10) << 4) | (v % 10)); }

HAL_StatusTypeDef ds3231_set_time(uint8_t h, uint8_t m, uint8_t s)
{
   uint8_t t[3] = { dec2bcd(s), dec2bcd(m), dec2bcd(h) };
   return HAL_I2C_Mem_Write(&hi2c1, DS3231_ADDR, 0x00,
                            I2C_MEMADD_SIZE_8BIT, t, 3, 100);
}

HAL_StatusTypeDef ds3231_get_time(uint8_t *h, uint8_t *m, uint8_t *s)
{
   uint8_t t[3];
   HAL_StatusTypeDef r = HAL_I2C_Mem_Read(&hi2c1, DS3231_ADDR, 0x00,
                                          I2C_MEMADD_SIZE_8BIT, t, 3, 100);
   if (r == HAL_OK) {
      *s = bcd2dec(t[0] & 0x7F);
      *m = bcd2dec(t[1]);
      *h = bcd2dec(t[2] & 0x3F);      /* 24-hour mode */
   }
   return r;
}
```

Use in main(): call ds3231_set_time once to set the time, then in the while loop call ds3231_get_time every second and print it. Mem_Read performs: S, 0xD0 (write), register 0x00, Sr, 0xD1 (read), three data bytes with ACK, ACK, NACK, then P.

6.6 Example 3: LCD1602 through the PCF8574 backpack

Below is a small driver written in 4-bit mode that sends each LCD byte as one four-byte I2C transaction. Change LCD_ADDR to 0x3F if the scanner reports that address.

```
#define LCD_ADDR  (0x27 << 1)
#define LCD_RS    0x01
#define LCD_EN    0x04
#define LCD_BL    0x08

static void lcd_nibble(uint8_t nib, uint8_t rs)       /* used only in init */
{
```

```
    uint8_t v = (uint8_t)((nib << 4) | LCD_BL | (rs ? LCD_RS : 0));
    uint8_t d[2] = { (uint8_t)(v | LCD_EN), v };
    HAL_I2C_Master_Transmit(&hi2c1, LCD_ADDR, d, 2, 50);
}

static void lcd_byte(uint8_t b, uint8_t rs)
{
    uint8_t hi = (uint8_t)((b & 0xF0) | LCD_BL | (rs ? LCD_RS : 0));
    uint8_t lo = (uint8_t)(((b << 4) & 0xF0) | LCD_BL | (rs ? LCD_RS : 0));
    uint8_t d[4] = { (uint8_t)(hi | LCD_EN), hi, (uint8_t)(lo | LCD_EN), lo };
    HAL_I2C_Master_Transmit(&hi2c1, LCD_ADDR, d, 4, 50);
}

void lcd_init(void)
{
    HAL_Delay(50);
    lcd_nibble(0x3, 0); HAL_Delay(5);
    lcd_nibble(0x3, 0); HAL_Delay(1);
    lcd_nibble(0x3, 0); HAL_Delay(1);
    lcd_nibble(0x2, 0); HAL_Delay(1);     /* switch to 4-bit mode */
    lcd_byte(0x28, 0);                    /* 4-bit, 2 lines, 5x8 font */
    lcd_byte(0x0C, 0);                    /* display on, cursor off */
    lcd_byte(0x01, 0); HAL_Delay(2);      /* clear display */
    lcd_byte(0x06, 0);                    /* cursor moves right */
}

void lcd_set_cursor(uint8_t row, uint8_t col)
{
    lcd_byte((uint8_t)(0x80 | (row ? 0x40 : 0x00) | col), 0);
}

void lcd_print(const char *s)
{
    while (*s) lcd_byte((uint8_t)*s++, 1);
}
```

Use in main(): lcd_init(); lcd_set_cursor(0,0); lcd_print("Hello, Uganda"); lcd_set_cursor(1,0); lcd_print("Tet la'ngech!"); This reproduces the message shown on the title slide. To show the clock, format the time with snprintf into a 17-character buffer and print it.

Why the enable pulse works: each PCF8574 byte takes tens of microseconds on the bus, much longer than the 450 ns minimum enable pulse of the HD44780, so no extra delay is needed between the E = 1 and E = 0 bytes. The clear and home commands need about 2 ms, which is why HAL_Delay(2) follows the clear command.

6.7 Example 4: interrupt mode

In interrupt mode the call returns immediately, and the program continues. Enable the I2C1 event and error interrupts in the NVIC settings first.

```
uint8_t rtc_buf[3];
volatile uint8_t rtc_ready = 0;
volatile uint32_t i2c_err = 0;

/* start a read in the main loop */
HAL_I2C_Mem_Read_IT(&hi2c1, DS3231_ADDR, 0x00, I2C_MEMADD_SIZE_8BIT,
rtc_buf, 3);

/* callbacks (put in USER CODE BEGIN 4) */
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
  if (hi2c->Instance == I2C1) rtc_ready = 1;
}

void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c)
{
  i2c_err = HAL_I2C_GetError(hi2c);
}

/* in while(1) */
if (rtc_ready) {
  rtc_ready = 0;
  /* convert rtc_buf with bcd2dec and display */
}
```

The callbacks are called from the interrupt service routine, so keep them short. Do not call HAL_Delay or blocking I2C functions inside them. Use a flag, as above, and do the work in the main loop. Start the next transfer only after the previous one has finished (hi2c1.State == HAL_I2C_STATE_READY), otherwise HAL returns HAL_BUSY.

6.8 Example 5: DMA mode

After adding the I2C1_TX and I2C1_RX DMA requests in CubeMX, the code looks the same as interrupt mode with _DMA in the function name:

```
HAL_I2C_Mem_Read_DMA(&hi2c1, DS3231_ADDR, 0x00,
I2C_MEMADD_SIZE_8BIT, rtc_buf, 3);
HAL_I2C_Master_Transmit_DMA(&hi2c1, LCD_ADDR, frame, frame_len);
```

The buffer must stay valid until the completion callback, so use a global or static array, never a local variable that goes out of scope. For a few bytes, DMA is not faster than polling; its advantage appears when sending hundreds or thousands of bytes, for example the 1024-byte frame of a 128x64 OLED, while the CPU does other work.

6.9 Error handling

- Always check the return value. HAL_OK means the slave acknowledged and all bytes moved.
- HAL_ERROR with HAL_I2C_GetError(&hi2c1) containing HAL_I2C_ERROR_AF means the address or a data byte was NACKed: wrong address, device not powered, or wrong shift of the address.
- HAL_TIMEOUT or HAL_BUSY that never clears means a line is stuck low (a slave holding SDA low after a reset in the middle of a byte) or the previous transfer never finished. Remedy: power-cycle the slave, or toggle SCL up to nine times as GPIO outputs until SDA is released, then re-initialise I2C1 (HAL_I2C_DeInit followed by MX_I2C1_Init).
- ARLO (arbitration lost) and BERR (bus error) point to noise, very long wires or two masters.

6.10 STM32 as an I2C slave (short note)

Set the "Primary slave address" in CubeMX Parameter Settings and use HAL_I2C_Slave_Receive and HAL_I2C_Slave_Transmit (or the _IT versions with HAL_I2C_AddrCallback). On the STM32F4 HAL the own address is stored in the 8-bit (shifted) form, so check the value in the generated MX_I2C1_Init(). This is used in Task 8 only if time allows.

7. I2C ON THE ARDUINO (WIRE LIBRARY)

7.1 Hardware

On the Arduino Uno, SDA is pin A4 and SCL is pin A5 (the two extra pins near AREF on Uno R3 are connected to the same lines). On the Mega, SDA is pin 20 and SCL is pin 21. The Uno runs at 5 V logic, unlike the 3.3 V STM32. The Wire library enables the microcontroller's weak internal pull-ups; add external pull-ups when the bus is long or fast. Include the library with #include <Wire.h>. The library uses the 7-bit address directly; do not shift it. The Wire buffer on AVR boards is 32 bytes, so send at most 32 bytes (including the register byte) in one transmission.

7.2 Wire library functions

Function	**Meaning**
Wire.begin()	Join the bus as master
Wire.begin(address)	Join the bus as slave with that 7-bit address
Wire.setClock(400000)	Change bus speed (default 100 kHz)
Wire.beginTransmission(addr)	Start building a write to that slave
Wire.write(byte)	Queue a byte (or array) to send
Wire.endTransmission([stop])	Send the queued bytes. true (default) sends STOP, false sends repeated START. Returns an error code

Function	Meaning
Wire.requestFrom(addr, count)	Read count bytes from a slave. Returns the number of bytes received
Wire.available()	Number of received bytes waiting
Wire.read()	Read one received byte
Wire.onReceive(handler), Wire.onRequest(handler)	Slave mode callbacks

Return codes of endTransmission():

Code	Meaning
0	Success
1	Data too long for the buffer
2	NACK received on transmitting the address (no such device)
3	NACK received on transmitting data
4	Other error
5	Timeout

7.3 Example 1: I2C scanner

```
#include <Wire.h>

void setup() {
  Serial.begin(9600);
  Wire.begin();
  Serial.println("I2C scanner");
}

void loop() {
  byte found = 0;
  for (byte addr = 8; addr < 120; addr++) {
    Wire.beginTransmission(addr);
    byte err = Wire.endTransmission();
    if (err == 0) {
      Serial.print("Found device at 0x");
      if (addr < 16) Serial.print("0");
      Serial.println(addr, HEX);
      found++;
    } else if (err == 4) {
      Serial.print("Unknown error at 0x");
      Serial.println(addr, HEX);
    }
  }
```

```
  if (found == 0) Serial.println("No devices found");
  Serial.println("Scan done");
  delay(3000);
}
```

7.4 Example 2: DS3231 clock (register read with repeated START)

```
#include <Wire.h>

const byte RTC = 0x68;

byte bcd2dec(byte v) { return (v >> 4) * 10 + (v & 0x0F); }
byte dec2bcd(byte v) { return ((v / 10) << 4) | (v % 10); }

void setTime(byte h, byte m, byte s) {
  Wire.beginTransmission(RTC);
  Wire.write(0x00);              // start at the seconds register
  Wire.write(dec2bcd(s));
  Wire.write(dec2bcd(m));
  Wire.write(dec2bcd(h));
  Wire.endTransmission();
}

bool getTime(byte &h, byte &m, byte &s) {
  Wire.beginTransmission(RTC);
  Wire.write(0x00);
  if (Wire.endTransmission(false) != 0) return false;   // repeated START
  if (Wire.requestFrom((uint8_t)RTC, (uint8_t)3) != 3) return false;
  s = bcd2dec(Wire.read() & 0x7F);
  m = bcd2dec(Wire.read());
  h = bcd2dec(Wire.read() & 0x3F);
  return true;
}

void setup() {
  Serial.begin(9600);
  Wire.begin();
  // setTime(12, 0, 0);          // run once, then comment out
}

void loop() {
  byte h, m, s;
  if (getTime(h, m, s)) {
    Serial.print(h); Serial.print(":"); Serial.print(m);
    Serial.print(":"); Serial.println(s);
  } else {
    Serial.println("RTC not responding");
  }
```

```
  delay(1000);
}
```

7.5 Example 3: LCD1602 I2C display

Install a LiquidCrystal_I2C library through the Library Manager (several versions exist; any that provides init(), backlight(), setCursor() and print() will do). The slide wiring is: SDA to A4, SCL to A5, VCC to 5 V, GND to GND.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);   // address, columns, rows (try 0x3F)

void setup() {
  lcd.init();
  lcd.backlight();
  lcd.setCursor(0, 0);
  lcd.print("Hello, Uganda");
  lcd.setCursor(0, 1);
  lcd.print("Tet la'ngech!");
}

void loop() { }
```

7.6 Example 4: OLED SSD1306 (outline)

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

Adafruit_SSD1306 display(128, 64, &Wire, -1);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);   // 7-bit address (0x78 >> 1)
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);
  display.println("Kyambogo");
  display.display();                           // send the frame buffer
}

void loop() { }
```

7.7 Example 5: Arduino as an I2C slave

```
#include <Wire.h>
```

```
volatile byte command = 0;

void receiveEvent(int howMany) {
  while (Wire.available()) command = Wire.read();
}

void requestEvent() {
  Wire.write(analogRead(A0) >> 2);   // send 8-bit value of the pot on A0
}

void setup() {
  Wire.begin(0x08);                 // slave address 0x08
  Wire.onReceive(receiveEvent);
  Wire.onRequest(requestEvent);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, command ? HIGH : LOW);
}
```

A master that writes 1 turns the LED on, writes 0 turns it off, and a master read returns the potentiometer value. The handlers run inside an interrupt, so keep them short.

8. STM32 AND ARDUINO SIDE BY SIDE

Topic	**STM32 (HAL)**	**Arduino (Wire)**
Address form	7-bit address shifted left by one (0x27 << 1)	7-bit address as it is (0x27)
Initialisation	CubeMX generates MX_I2C1_Init(); handle hi2c1	Wire.begin()
Pins	I2C1: PB8 SCL, PB9 SDA (or PB6, PB7)	Uno: A5 SCL, A4 SDA
Logic level	3.3 V	5 V on Uno
Speed setting	CubeMX parameter, 100000 or 400000 Hz	Wire.setClock(400000)
Write bytes	HAL_I2C_Master_Transmit()	beginTransmission, write, endTransmission
Read bytes	HAL_I2C_Master_Receive()	requestFrom, available, read
Register read	HAL_I2C_Mem_Read() (repeated START inside)	endTransmission(false) then requestFrom
Device present?	HAL_I2C_IsDeviceReady()	endTransmission() returns 0

Topic	STM32 (HAL)	Arduino (Wire)
Error information	HAL_StatusTypeDef and HAL_I2C_GetError()	endTransmission() return code 1 to 5
Interrupt, DMA	_IT and _DMA functions with callbacks	Not for master transfers (blocking); slave uses onReceive and onRequest
Speed of development	More set-up, finer control	Very fast to start, less control

9. TROUBLESHOOTING GUIDE

Symptom	Likely cause	What to do
Scanner finds nothing	Wiring, no power, SDA and SCL swapped, no common ground, no pull-ups	Check wiring and supply; measure SDA and SCL idle at about VDD; add 4.7 kohm pull-ups
STM32 gets HAL_ERROR (AF flag), Arduino code 2	Wrong address, or forgot << 1 on STM32	Run the scanner; compare 7-bit and 8-bit forms
Arduino returns code 3	Slave accepted the address but rejected data	Check register and command values; check the datasheet
LCD lit but blank or black blocks	Contrast trimmer, wrong address, initialisation not done	Turn the trimmer; verify with scanner; run init
LCD shows garbage	Nibble order or RS and E bits wrong in driver; noise	Compare masks 0x01, 0x04, 0x08 and the pin map of your backpack
Works at 100 kHz, fails at 400 kHz	Rise time too slow (pull-up too large or cable too long)	Reduce Rp, shorten wires, check with a scope
Bus hangs, SDA stuck low	A slave interrupted in the middle of a byte	Power-cycle slave; clock SCL up to nine times; re-initialise I2C
Two modules never both work	Same address	Change address pins or jumpers; use a multiplexer
Rounded or sluggish edges on the scope	Large bus capacitance	Smaller Rp, fewer devices, shorter wires
STM32 module works, Arduino 5 V module corrupts	Voltage mismatch	Use a level shifter or 3.3 V modules

11. STUDENT TASKS

Work in groups of three or four. Each group keeps one lab notebook and submits one short report per task: the circuit or wiring description, the code (commented), the observed results (serial monitor output, screenshots or photographs) and brief answers to the questions. Suggested marks are given in brackets, out of 100 for Tasks 1 to 8. Task 9 is a separate coursework project.

Equipment per group: STM32 Nucleo-F411RE (or the lab trainer with STM32F411), Arduino Uno, breadboard and jumper wires, LCD1602 with I2C backpack, DS3231 RTC module, SSD1306 OLED (optional), 4.7 kohm and 2.2 kohm resistors, oscilloscope or logic analyser, USB cables, bidirectional level shifter (for Task 7).

Task 1: Paper exercises

1. Convert to the other form and give the write and read address bytes: (a) 7-bit address 0x50; (b) 7-bit address 0x3F; (c) address byte 0xD1; (d) address byte 0x78 (write); (e) 7-bit address 0110000 from the slide.
2. Draw the complete bus frame (S, address, R/W, ACK, data, ACK, P) for writing the byte 0x55 to a device at 7-bit address 0x48. Mark the value of SDA at each SCL pulse and give the address byte in hexadecimal and binary.
3. A bus at 5 V has a total capacitance of 150 pF. Calculate the minimum and maximum pull-up resistor for (a) 100 kHz and (b) 400 kHz, and choose a standard value for each case. Use tr(max) = 1000 ns and 300 ns and a sink current of 3 mA at 0.4 V.
4. A sensor register-read of six data bytes uses the combined transaction S, addr+W, register, Sr, addr+R, six data bytes, P. Estimate the time it takes at 100 kHz and at 400 kHz, ignoring START and STOP overhead.
5. Why does the master send NACK after the last byte it reads, and not ACK?
6. Why can two devices with the same address not share a bus? List two ways of solving the problem.
7. Explain why I2C uses open-drain outputs and what would happen if two devices drove the lines with push-pull outputs.
8. Compare polling, interrupt and DMA modes of the STM32 HAL I2C driver. When would you choose each?

Task 2: Bus scanner on both platforms (10 marks)

Aim: discover which addresses respond on the bus.

1. Wire the LCD backpack and the DS3231 to SDA and SCL of the Arduino. Upload the scanner of Section 7.3. Record the addresses found.
2. Move the same modules to PB9 (SDA) and PB8 (SCL) of the STM32. Create the CubeMX project as in Section 6.2 and implement the scanner of Section 6.4. Record the addresses found.

3. Disconnect the RTC and rescan. Disconnect SDA only and rescan. Describe what you see in each case.
4. Questions: Do the Arduino and STM32 give the same list? Which addresses does each platform scan, and why do we avoid 0x00 to 0x07 and 0x78 to 0x7F? Convert each address found into its write and read byte.

Task 3: Hello LCD

1. Display “Hello, Uganda” on line 1 and your group name on line 2, on the Arduino (Section 7.5) and on the STM32 (Section 6.6).
2. Change the I2C address in the code to a wrong value (for example 0x28). Record what happens on each platform and the return value of the transmit function.
3. Modify the program to scroll a message longer than 16 characters across the display.
4. Question: the STM32 driver sends four bytes per LCD character in one transaction. How many bits are on the wire per character at 100 kHz and how long does it take? How many characters per second can the display be refreshed at most?

Task 4: Digital clock using two devices on one bus

1. Connect both the LCD (0x27) and the DS3231 (0x68) to the same SDA and SCL lines. Set the time once from your code.
2. Show the time as HH:MM:SS on line 1 and the date or temperature on line 2. The DS3231 temperature is in registers 0x11 (signed integer part) and 0x12 (upper two bits, in steps of 0.25 degrees Celsius).
3. Implement it on the Arduino and on the STM32 (HAL_I2C_Mem_Read and HAL_I2C_Mem_Write).
4. Add error handling: if the RTC does not answer, display “RTC ERROR” and keep retrying.
5. Questions: Why can two devices share the lines? What stops them from talking at the same time? What is BCD, and what would be displayed if you forgot to convert it?

Task 5: Observing the bus

Use an oscilloscope (two channels) or a logic analyser with an I2C decoder.

1. Capture SDA and SCL during one write to the LCD. Identify START, the address byte, the R/W bit, the ACK bit, the data byte and STOP. Sketch the capture and label it, and decode the address byte into hexadecimal.
2. Measure the clock frequency at 100 kHz and at 400 kHz settings. How close is the measured value to the setting?
3. Rise-time experiment: remove the module pull-ups (or use a bare bus) and fit external resistors of 1 kohm, 2.2 kohm, 4.7 kohm and 10 kohm in turn. Measure the 30 % to 70

% rise time of SDA for each. Tabulate the results and compare with tr = 0.8473 x Rp x Cb. Estimate Cb of your bus.

4. At 400 kHz, find the largest pull-up for which communication stays reliable. Does it agree with the calculation in Section 4.3?
5. Question: why is the rising edge slow but the falling edge fast?

Task 6: Fault finding

A group introduces one fault at a time while the others find it using only the scanner, return codes and the scope. Record in a table: the symptom, the return code or HAL error, how it was found and the fix.

- Wrong address (for example 0x3F instead of 0x27).
- Address not shifted on the STM32 (0x27 passed instead of 0x27 << 1).
- SDA and SCL swapped.
- One pull-up removed, then both removed.
- Two modules at the same address (set the same address jumpers on two PCF8574 backpacks).
- Module powered with 5 V pull-ups connected to the 3.3 V STM32 (lecturer demonstration only, using a multimeter; do not connect without confirming the pin tolerance).

Task 7: STM32 master, Arduino slave

1. Load the slave sketch of Section 7.7 on the Arduino (address 0x08) with a potentiometer on A0 and the built-in LED.
2. Connect the STM32 as master: SDA to SDA, SCL to SCL, common ground. Because the Uno is a 5 V device and the STM32 is 3.3 V, use a bidirectional level shifter between the two boards (or a 3.3 V Arduino). Make sure only one set of pull-ups to the correct voltage is used.
3. On the STM32 send 1 and 0 with HAL_I2C_Master_Transmit to switch the Arduino LED, using the address 0x08 << 1. Add a user-button toggle.
4. Read the potentiometer value with HAL_I2C_Master_Receive and display it on the LCD every 200 ms as a number and as a bar of up to 16 characters.
5. Questions: which side stretches the clock, if any? What happens if the Arduino is switched off (give the HAL status)? Draw the bus frames of the write and of the read.

Task 8: Interrupt and DMA modes, and timing

1. Rewrite the clock read of Task 4 using HAL_I2C_Mem_Read_IT and the completion callback of Section 6.7, and then using HAL_I2C_Mem_Read_DMA.

2. Toggle a GPIO pin before and after the call and measure on the scope how long the function takes to return in polling, interrupt and DMA modes. Also measure the total length of the transfer on the bus. Tabulate and explain.
3. Run the bus at 400 kHz and repeat. State the improvement and the limits to it.
4. Extension (optional): configure a second STM32 or the same board's I2C2 as a slave and communicate between I2C1 (master) and I2C2 (slave) on one board.

Task 9: Mini-project (separate course work)

Choose one project. Deliverables: working demonstration, wiring diagram, commented code, a 4 to 6 page report (design, pull-up calculation, test results) and a short presentation.

- Alarm clock: DS3231 time, LCD display, two buttons to set the alarm, buzzer.
- Temperature and time logger: read the DS3231 temperature every minute, show on the LCD and the OLED and send it to the PC over the UART.
- Two-board system: an STM32 master collects values from two Arduino slaves (addresses 0x08 and 0x09), each reading a different sensor, and displays them on the LCD.
- Analogue to I2C bridge: STM32 reads the ADC and the Arduino shows the values on an OLED over I2C.

Marking guide for Task 9: working system 40 %, code quality and comments 20 %, correct pull-up design and measurements 15 %, report 15 %, presentation and questions 10 %.