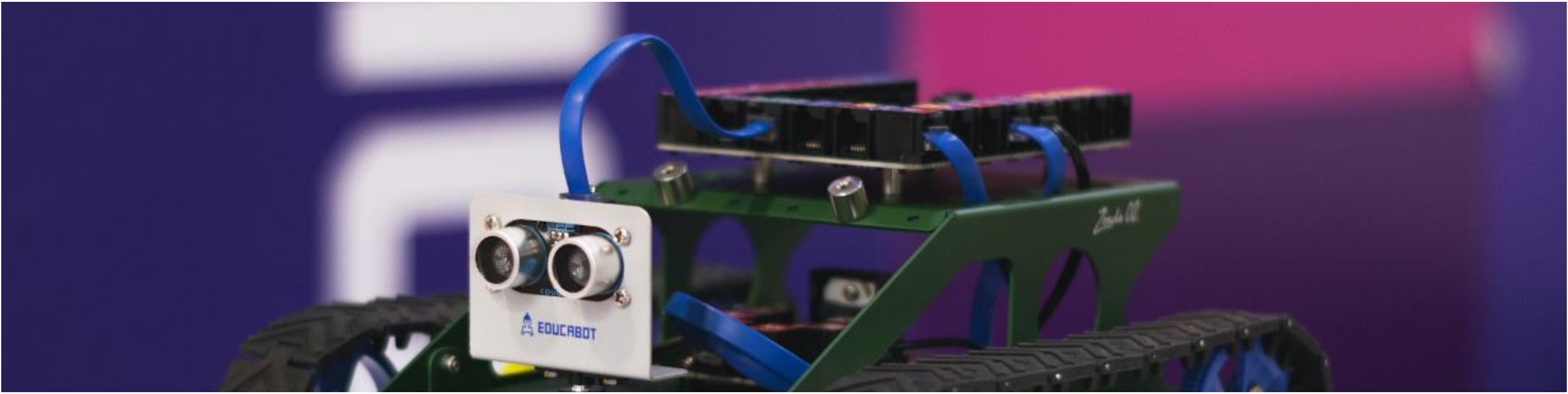


Arduino SYS-STEM for Schools

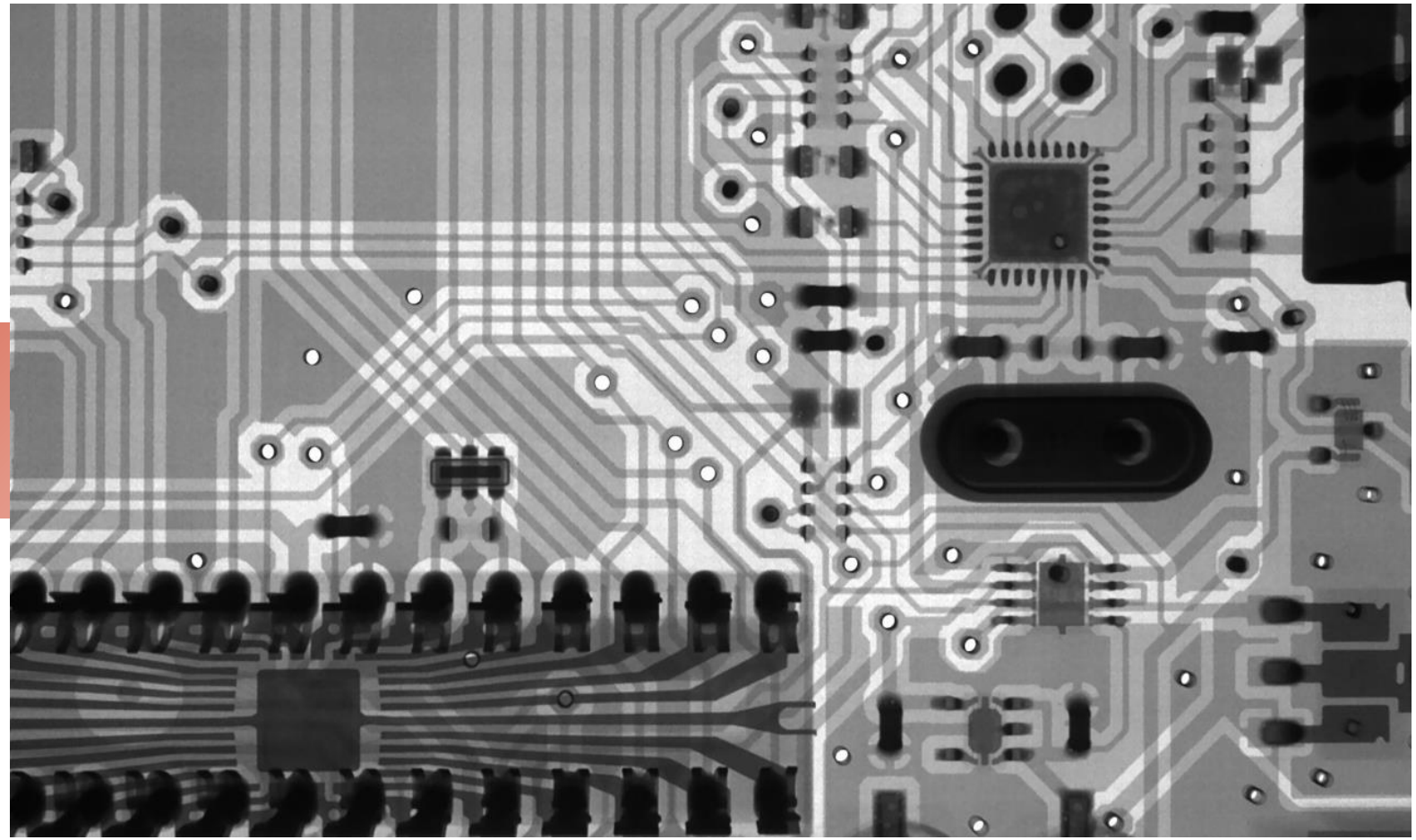
Training Methodology

SYS-STEM – Arduino SYS-STEM for Schools
Erasmus + Key Action 2 Strategic partnership - 2019-1-ES01-KA201-064454

MODULE 6

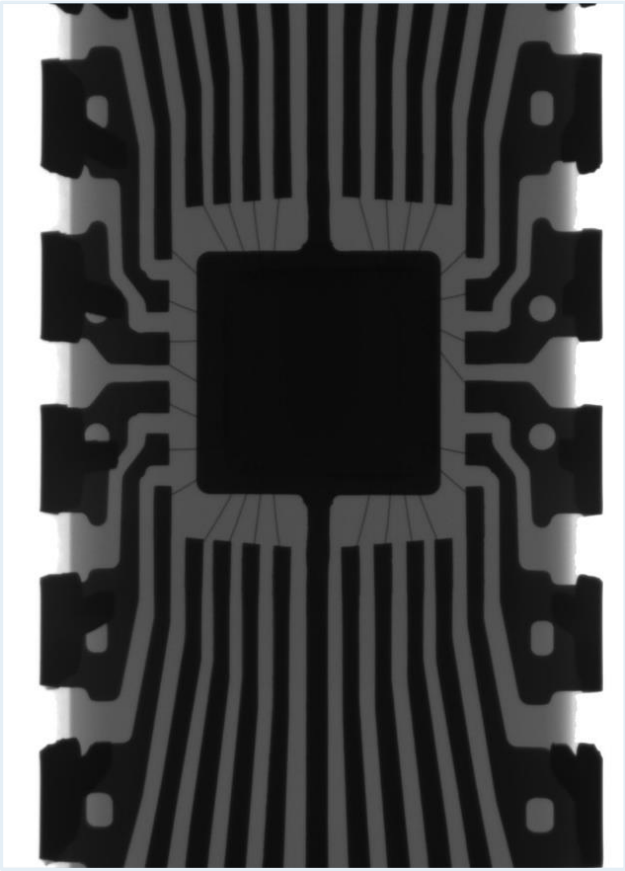
Liquid Crystal Displays

(utilizing the I2C serial protocol)



TRAINING MODULE CONTENTS

- Objective
- Learning Outcomes
- Unit 1 - ABOUT LCD DISPLAYS
- Unit 2 - PARALLEL & SERIAL COMMUNICATION
- Unit 3 - SOFTWARE CONSIDERATIONS
- Additional Reading Materials
- Exercises



OBJECTIVE

Module 6 will give a thorough overview of LCD displays, their different types, the way they work with Arduino's default and add-on libraries and how to prototype with them;

Moreover, it will help the understanding of the key differences between parallel and serial communication, enabling the selection of the best suited display, fit for a particular purpose.

EXPECTED LEARNING OUTCOMES

Knowledge

Upon completion of this module, the student will be able to:

- Understand the key differences in the way parallel and I2C displays work
- Know the pros and cons for each type of display and understand the optimal type to employ in each and every situation

Competences and Skills

Upon completion of this module, the student will be able to:

- Use an I2C-based serial LCD display, making use of most of the features provided by the LCD library

UNIT 1

ABOUT LCD DISPLAYS

ABOUT LCD DISPLAYS

- An LCD display is an output peripheral that can be used to display most kind of data, including alphanumerical characters, symbols and even simple graphics

- They're distinguished in two major types; the more complex to use "matrix" displays and the simpler devices organized in rows and columns. On the right, you can see a sample of the later, that is, a 16 columns x 2 rows LCD display

The most common LCD display on the market; a 16x2 (cols x rows), based on the HD44780 controller by Hitachi

PREDEFINED CHARACTERS

- There is an internal ROM memory integrated into the LCD display's controller (which in most displays, is a variant of the Hitachi HD44780) that hosts character definitions.

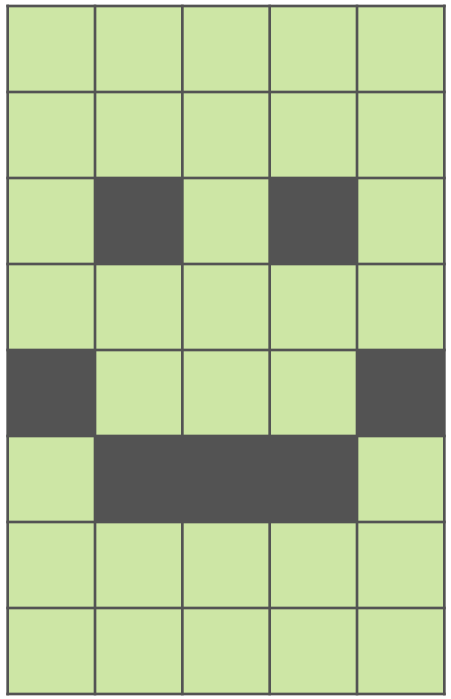
- Most common characters (like the latin charset and some basic symbols) are already there, defined and ready to use

The most common character codes that are pre-defined in the HD44780 controller

CUSTOM CHARACTERS

- You can define custom characters to use with your programs; however, there is a limit of 8 total custom characters

- Each character (or "glyph") is represented through a 5x8 matrix of pixels. These are defined in code, but there is also a number of online tools that allow the easy creation of custom characters.



A "smiley" character coded in a 5x8 fashion, as per the display's requirements

UNIT 2

PARALLEL & SERIAL COMMUNICATION

PARALLEL COMMUNICATION (INTRO)

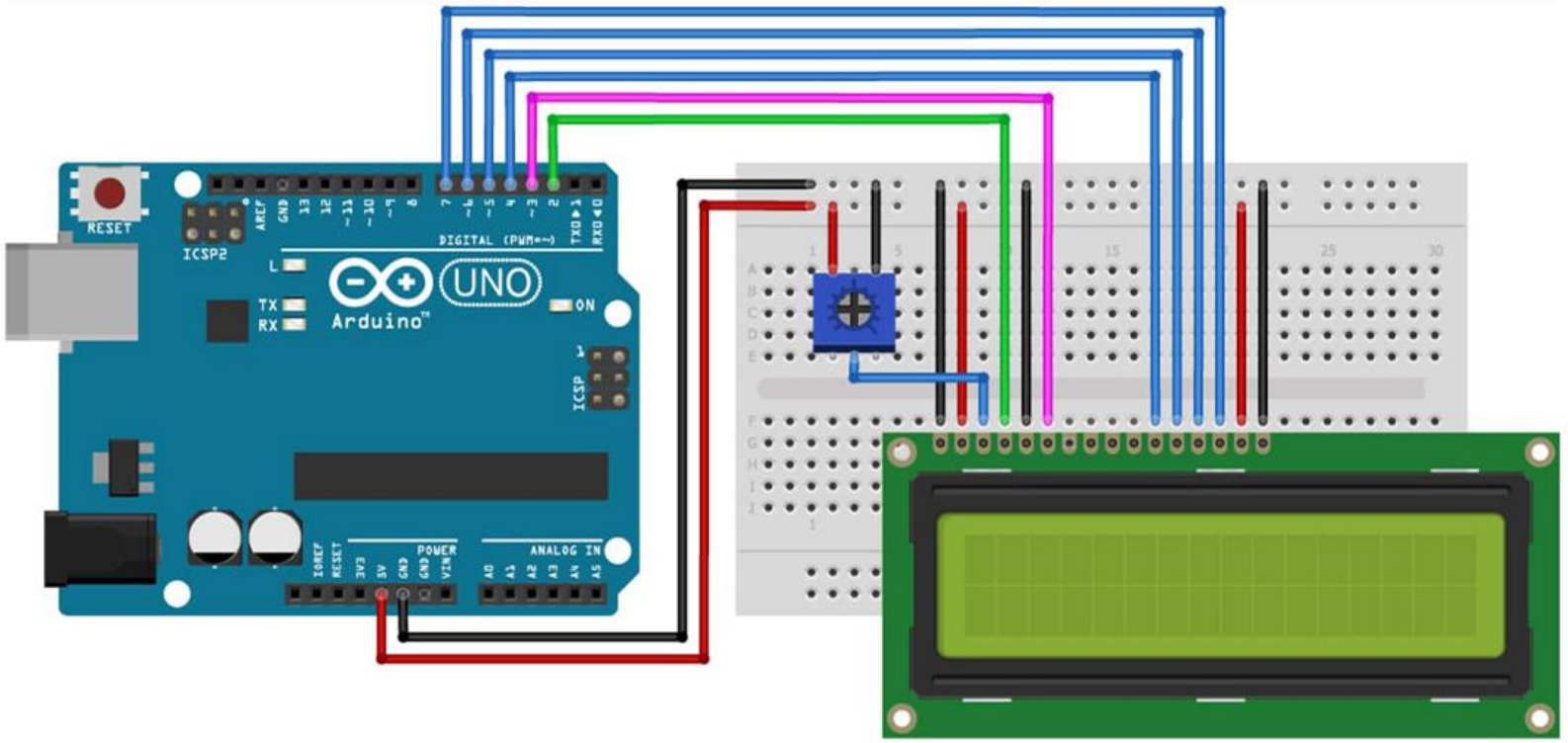
- Parallel communication employs a larger number of pins, resulting in a somewhat complex circuit, but allows for fast and easy data transfer between the LCD display and the microcontroller

- The pins, when working in parallel mode, are divided into three groups;
  - Power supply pins
  - Control pins
  - Data pins

PARALLEL COMMUNICATION (PINS)

PIN#	NAME	TYPE	DESCRIPTION
1	**VSS / GND**	POWER SUPPLY	Power supply ground
2	**VDD / VCC**	POWER SUPPLY	Power supply positive rail (usually +5V)
3	**V0**	POWER SUPPLY	Variable voltage between VSS and VDD for contrast adjustment
4	**RS**	INPUT	Output from the controller (LOW or HIGH) switches between transferring data, or instructions
5	**R/W**	INPUT	Output from the controller (LOW or HIGH) switches between writing data to the LCD, or reading from it
6	**E**	INPUT	Output from the controller (LOW or HIGH) switches between the display being disabled, or not
7...14	**DB0...DB7**	INPUT/OUTPUT	Data and instruction bus lines
15	**L+**	POWER SUPPLY	Power supply positive rail (usually +5V) for the LED backlight
16	**L-**	POWER SUPPLY	Power supply ground for the LED backlight

PARALLEL COMMUNICATION (CIRCUIT)



SERIAL (I2C) COMMUNICATION

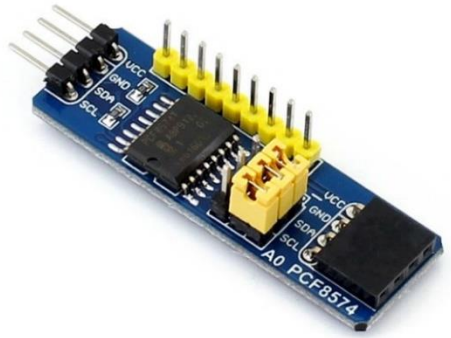
- Serial communication, on the other hand, employs a much smaller number of pins. We'll be focusing on I2C, which only requires 2 pins (in addition to power and ground). Other protocols (like SPI) would require a different number of pins. This allows for a much simpler circuit which is easier to implement

- On the downside, the code can become much more complex and harder to maintain due to the added layer of functionality. Luckily for us, there are libraries available that abstract away the complexity

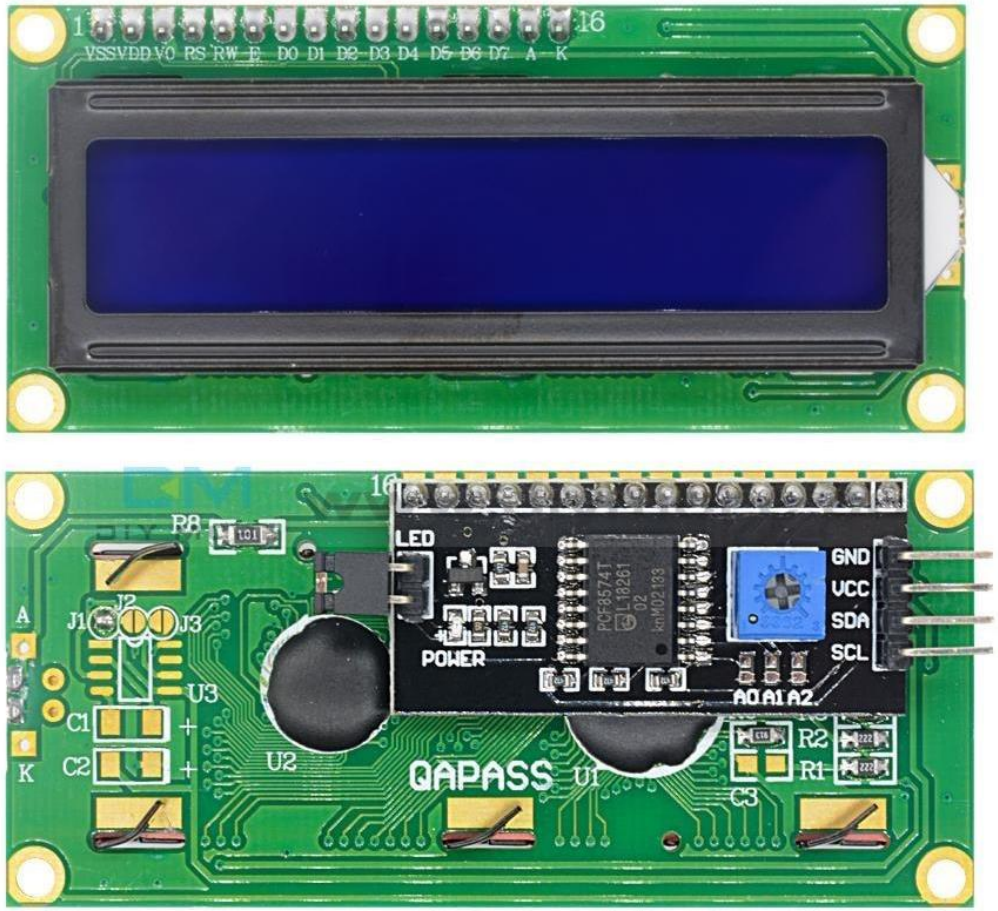
SERIAL I/O EXPANDERS

- A number of ICs exist, like the well-established in the Arduino world PCF8574 that work as I2C-based I/O expanders, enabling us to use an 8-bit parallel interfaces over I2C

- Of course, there is a great variety of LCD displays and add-on modules that already feature these I/O expanders, which frees us from the need to design and implement a circuit ourselves



A well-known I2C-based I/O expander module, built on the common PCF8574 which can be used alongside with 8-bit parallel LCD displays



LCD DISPLAY W/ I2C ADD-ON

On the left, a parallel LCD display is "capped" with an I2C I/O expander, allowing us to issue commands and send data through I2C.

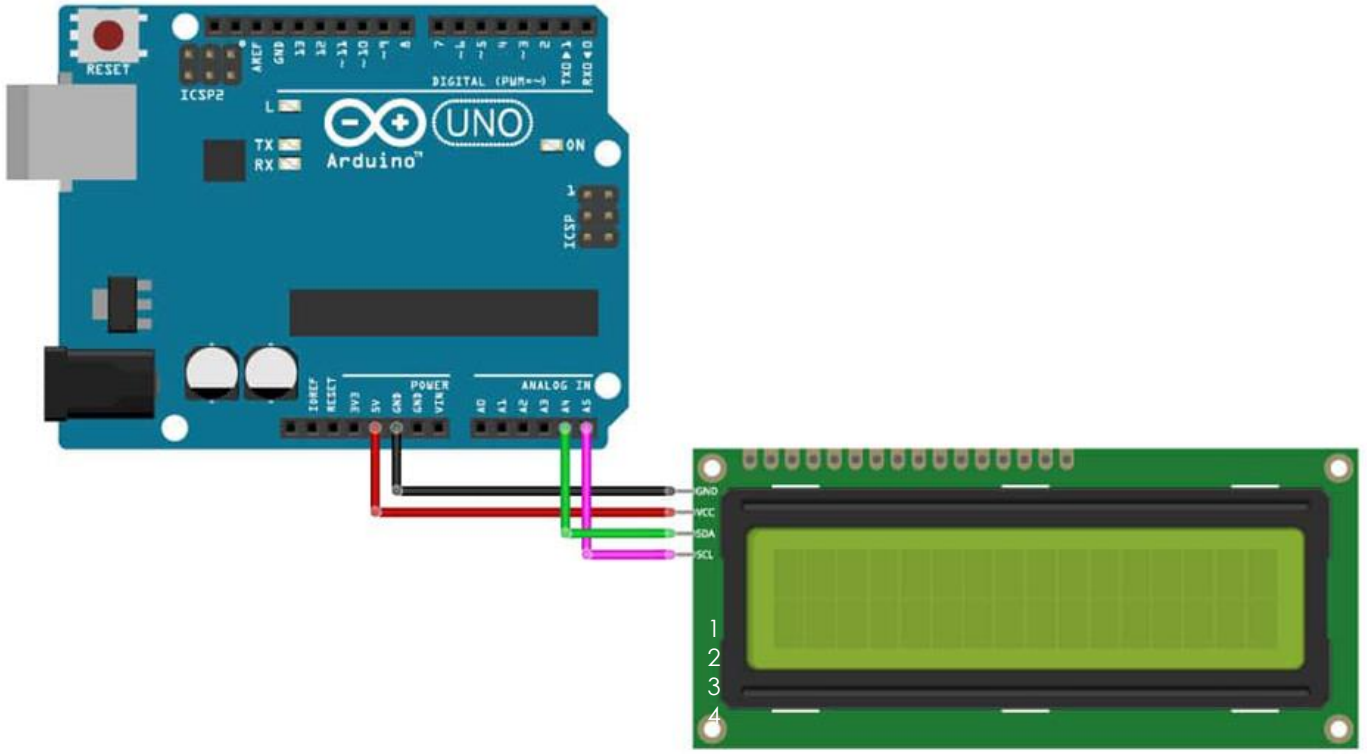
Of course, LCD displays that implement I2C natively do exist, but these are more expensive. As of that, they are less common for hobbyist or educational use.

I2C COMMUNICATION (PINS)

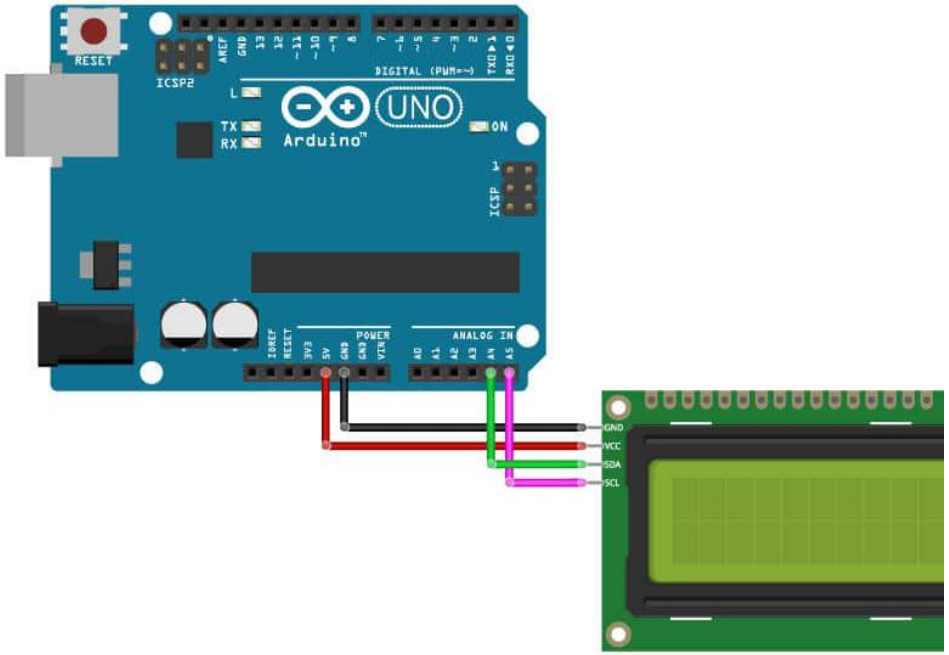
PIN#	NAME	TYPE	DESCRIPTION
1	**VSS / GND**	POWER SUPPLY	Power supply ground
2	**VDD / VCC**	POWER SUPPLY	Power supply positive rail (usually +5V)
3	**SDA**	INPUT/OUTPUT	Serial data
4	**SCL**	CLOCK	Serial clock. Communication is timed by the master device (e.g the Arduino)

See the difference? Apart from power and ground, communicating via I2C requires only 2 pins, from which only 1 (SDA) is actual data; the SCL line carries timed clock pulses to synchronise the transmission of data.

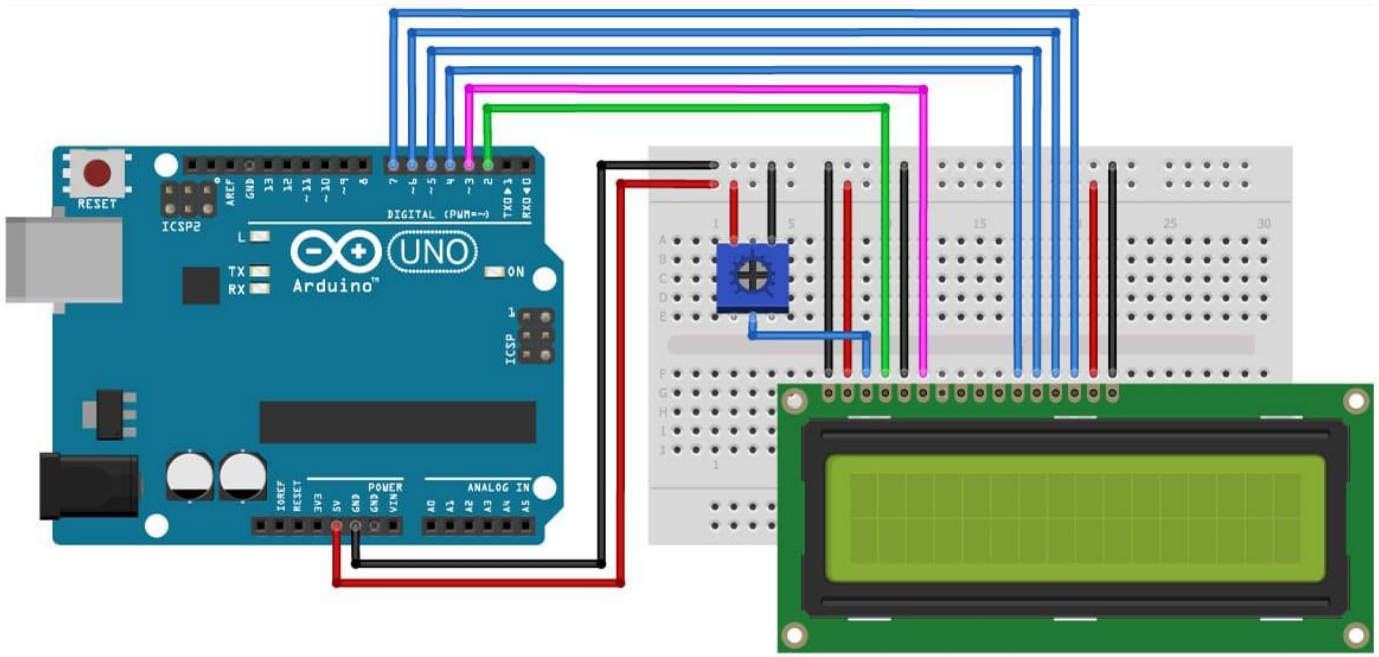
I2C COMMUNICATION (CIRCUIT)



PARALLEL VS SERIAL



A serial prototype using I2C. The circuit is much simpler than the parallel one.



A 4-bit parallel prototype. The missing 4 bits eliminate the need for 4 more wires, but make communication slower

PARALLEL VS SERIAL (PROS & CONS)

- Parallel is
  - Faster in terms of data transfer and execution on the microcontroller, whilst it needs less coding by the developer (in LOC) and is cheaper to implement in circuit (requires less and/or cheaper components);
  - However, it may result in a complex *prototype* and may require greater development time, leading to lengthier time to market
- Serial (I2C) on the other hand, is
  - Easier to deploy especially for non-experts;
  - Slower in terms of data transfer and execution, ad may result in a larger codebase and is generally more expensive;
  - However, helping the developer avoid circuit mishaps, it eases prototyping and allows for less development time

UNIT 3

SOFTWARE CONSIDERATIONS

LCD LIBRARIES (1)

- Before being able to communicate with an I2C LCD display, we need to be able to communicate via I2C in general.

- Arduino provides the so-called <Wire.h> library (I2C library) that frees us from the need to 'talk' directly to the microcontroller in low-level. However, using the I2C library is also cumbersome. Although it abstracts away the complexity of talking low-level, we would still need to mess with the LCD's registers, issuing commands and writing bytes.

LCD LIBRARIES (2)

- The complexity of talking to LCD display via I2C is simplified by the available library. It provides a friendlier and abstract way to communicate with the display itself, rather that writing to its' registers.

- The most common library for I2C displays is the following;

  - The `<LiquidCrystal_I2C.h>` library, developed by *Frank de Brabander*, available on GitHub and/or GitLab (at the time of writing) (https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library).

LCD LIBRARIES (3)

- Mind that, there is a plethora of libraries available, either for parallel or for I2C, each one targeting specific uses and having its own pros and cons. There is no "good" or "bad" library; each one is simply fit for a particular purpose
  - In this unit we will focus on the I2C-based library by *Frank de Brabander*. It is the library that the exercises are based and tested on, and it is a great general purpose I2C-based LCD library

- From now on, we are going to dive into studying the most representative and important functions and aspects of the aforementioned library

INCLUDING THE LIBRARY

- The library needs to be installed first. Download the compressed ZIP file from GitHub/GitLab and install it in your Arduino IDE distribution by selecting "`Sketch > Include Library > Add .ZIP Library...`“ from the menu. The Wire (I2C) library is already included.

- Both libraries need to be included in your program before they can be utilized. Issue the following compiler instructions in the top section of your code where most of your “include” statements reside.

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

CONSTRUCTOR: LiquidCrystal_I2C()

- This is the constructor; it creates a "LiquidCrystal_I2C" type of object, which will be used to interact with the display. It is issued like this;

```
LiquidCrystal_I2C LCD([addr], [cols], [rows])
```

NAME	DESCRIPTION
LCD	The name of the constructor. Later on, we will use this to refer to the object.
[addr]	The I2C address of the display or I/O expander. You will need to look up this in the datasheet, but the most common values are 0x3F and 0x27.
[cols]	The number of columns. In our case is 16.
[rows]	The number of rows. In our case is 2.

METHOD: LCD.init()

- This method initialises communication between the Arduino and the slave I2C device (the LCD). It requires no arguments.

```
LCD.init();
```

METHODS: LCD.backlight(), LCD.noBacklight()

- This method enables the LCD's backlight. Since we are now communicating through I2C and we do not have direct access to the LCD's backlight power pins, we need an indirect method of enabling it. The method requires no arguments, but the default state of the LCD is to have its' backlight turned on.

```
LCD.backlight();
```

- There is also a separate method to switch off the LCD's backlight if desired, which is issued as follows;

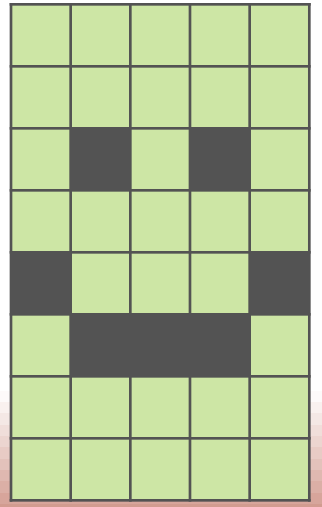
```
LCD.noBacklight();
```

METHOD: LCD.createChar()

- This method creates a custom character and stores it in the display's memory. A total of 8 glyphs can be defined, and each one is represented using a 5x8 matrix of pixels, like so;

```
byte smiley[8] = { B00000, B00000, B01010, B00000, B10001, B01110, B00000, B00000 };
LCD.createChar(0, smiley);
```

- The above code would produce the smiley shown on the right, but it would NOT print it, rather just store it in the LCD's memory for later use



METHODS: LCD.home(), LCD.clear()

- This method "homes" the cursor, positioning it in the top left corner (column 0, row 0) of the screen. It needs no arguments. It does NOT clear the contents, it only changes the cursor's position.

```
LCD.home();
```

- There is also a separate method to home and clear the LCD, simultaneously.

```
LCD.clear();
```

METHOD: LCD.setCursor()

- This method positions the cursor (i.e the position that new text will be placed) at the requested place. The cursor can be either visible or invisible (more on that later)

```
LCD.setCursor([col], [row]);
```

NAME	DESCRIPTION
[col]	The desired column (0...15) to position the cursor
[row]	The desired row (0...2) to position the cursor

METHODS: LCD.print(), LCD.write()

- The first method simply prints some data to the LCD display, at the cursor's position;

```
LCD.print([data], [base]);
```

NAME	DESCRIPTION
[data]	The desired data to print
[base]	The base in which to print numbers. Possible options are BIN, OCT, DEC, HEX. This parameter is optional

- On the other hand, LCD.write() will output a single character on the display, enabling us to print ASCII codes with ease. For example, the following statement will output the degree symbol (°) on the LCD;

```
LCD.write(0xDF);
```

METHODS: LCD.cursor(), LCD.noCursor()

- As said, the cursor can either be visible, or hidden. This is controlled with these two methods, namely cursor() and noCursor(), which either show, or hide the cursor. Both of them expect no arguments.

```
LCD.cursor();
LCD.noCursor();
```

The cursor will be displayed either as an underscore, or as a square-ish block. This depends upon the LCD display's controller implementation. By default, the cursor will not be displayed.

METHODS: LCD.blink(), LCD.noBlick()

- In a similar manner, we can also blink (or stop the blinking of) the cursor. Both of these methods expect no arguments.

```
LCD.blink();
LCD.noBlink();
```

By default, the cursor will not blink.

METHODS: LCD.display(), LCD.noDisplay()

- The LCD display can also switch-off, but without losing its' data. Neat. You can use noDisplay() to shut-down the display, and display() to power it up again, at the same time retrieving its' data and the cursor's position.

```
LCD.display();
LCD.noDisplay();
```

METHODS: LCD.scrollDisplayLeft(), LCD.scrollDisplayRight()

- Use these two methods to scroll the display. Each one will displace any existing content (including the cursor) on the LCD one place to the left, or to the right, respectively. They expect no arguments, but it is important to remember that the whole LCD will be scrolled and not a single line of it.

```
LCD.scrollDisplayLeft();

LCD.scrollDisplayRight();
```

METHODS: LCD.leftToRight(), LCD.rightToLeft()

- Use these two methods change the direction the text is printed on the LCD. By default, the LCD will print from left to right, but you can use rightToLeft() to reverse that. Both expect no arguments.

```
LCD.leftToRight();

LCD.rightToLeft();
```

METHODS: LCD.autoscroll(), LCD.noAutoscroll()

- The first method, will make the text sent to the LCD to be printed at the same position, forcing any previous text to be "pushed" to the left or right (respecting the leftToRight() and rightToLeft() methods). By default, autoscroll is disabled.
- The second method disables autoscroll, restoring the default behaviour

```
LCD.autoscroll();

LCD.noAutoscroll();
```

EXTRA READING MATERIALS

Online Resources

- Reference for the LiquidCrystal library by arduino.cc. An invaluable resource that applies not only to 8-bit parallel displays for which it has been designed, but also on I2C ones, as most third-party libraries are made compatible with it;
  - https://www.arduino.cc/en/Reference/LiquidCrystal

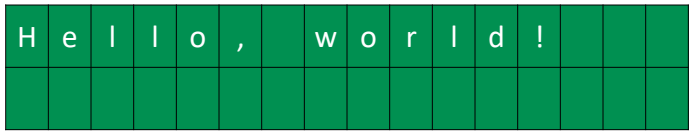
Online Resources

- The GitHub repository for the LiquidCrystal_I2C library that we're using for this module. Unfortunately, at the time of writing it didn’t contain a usage guide, but the examples are pretty much self-explanatory;
  - https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library

EXERCISES / TESTS / QUIZZES

EXERCISE 1 (INTRO)

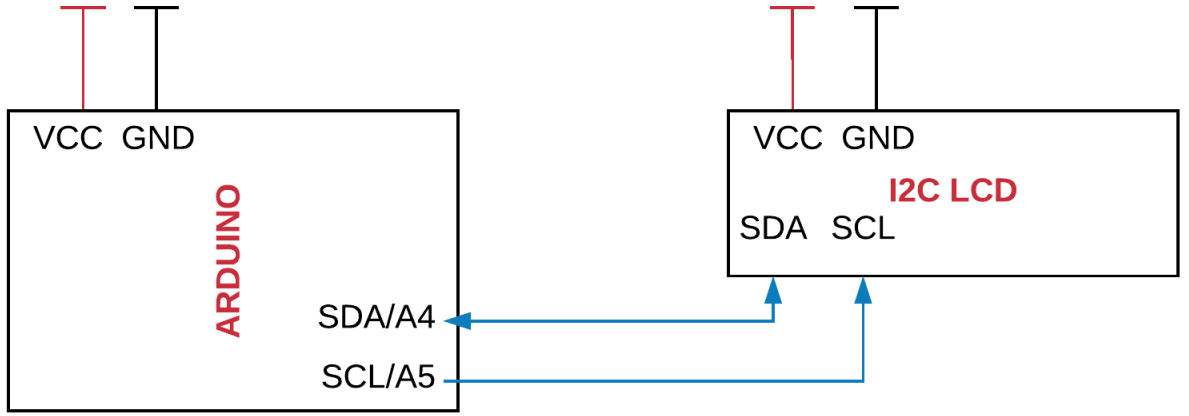
- This is an easy, introductory exercise.



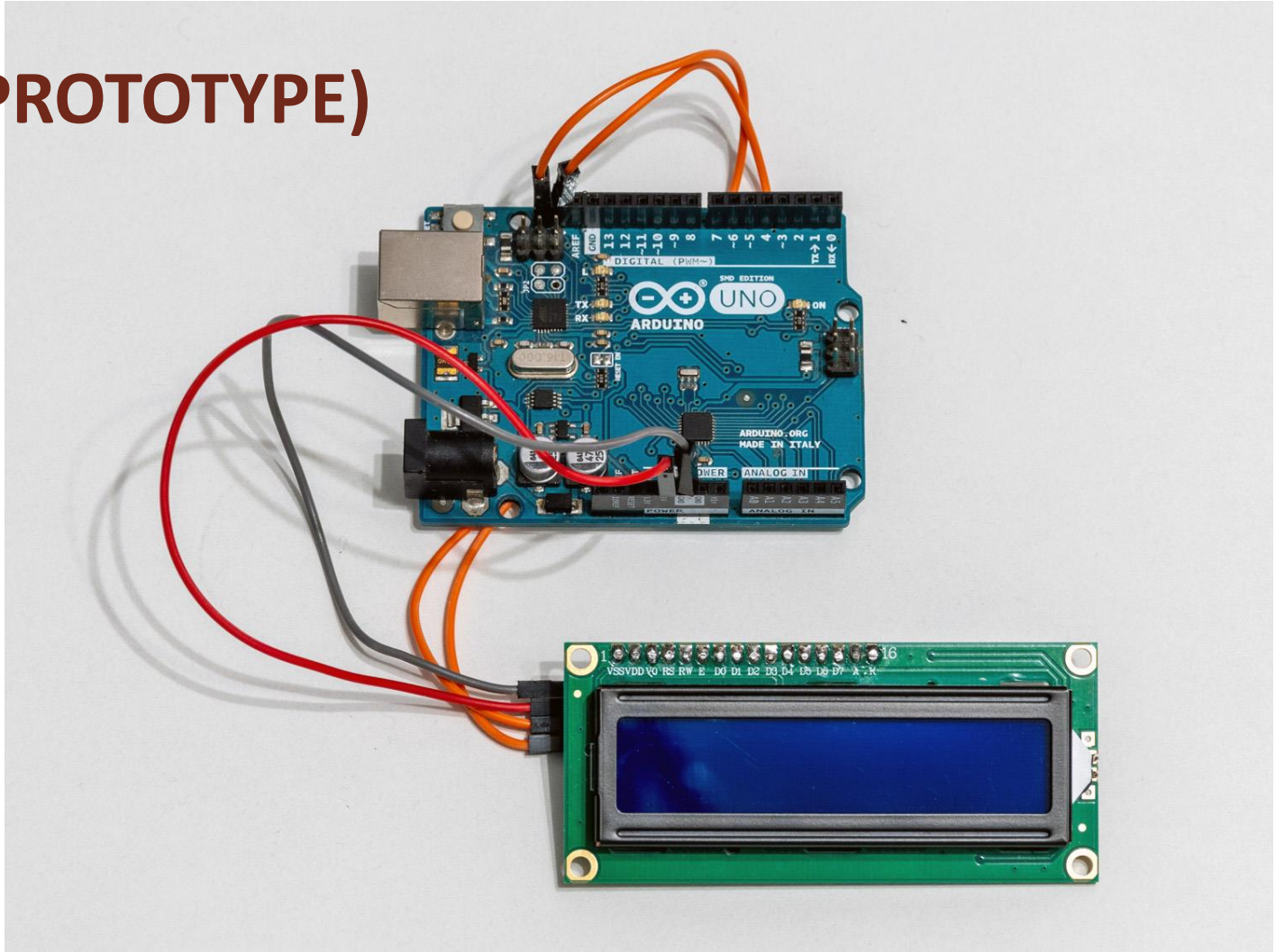
- A predefined "Hello, world!" text shall be displayed on the LCD display for a set duration, then hidden for the same amount of time, then shown again, etc. This process is to be repeated indefinitely.

- You are not allowed to use delay() or delayMicroseconds() in your code, thus, you need to figure a way to achieve the same result using the timekeeping functions.

EXERCISE 1 (SCHEMATIC)



EXERCISE 1 (PROTOTYPE)

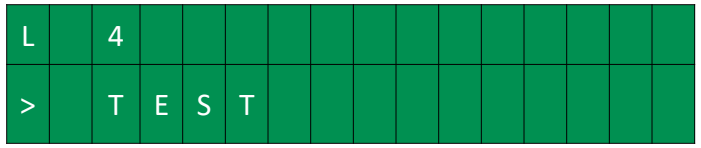


EXERCISE 1 (CODE)

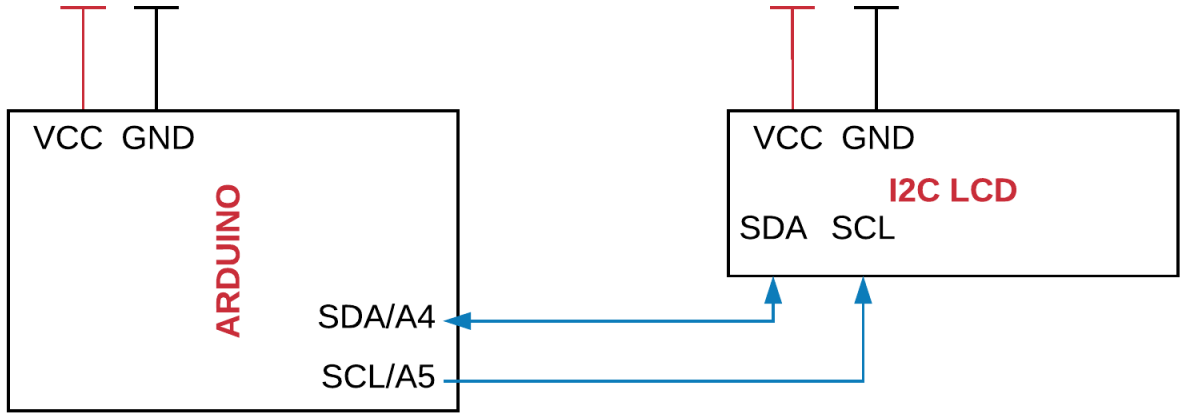
```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Defining an object named LCD of type LiquidCrystal_I2C
LiquidCrystal_I2C LCD(0x3F, 16, 2);

// Last is a timer variable
// Tick is used to invert to operation of our program
unsigned long last = 0, tick = 0;

void setup()
{
  // Initialises the LCD object
  LCD.init();

  // Turns on the backlight
  LCD.backlight();
}

void loop()
{
  // Run the program every 2000ms (2s):
  if (millis() > last + 2000)
  {
     // On even ticks, clear the LCD:
    if ( tick % 2 ) {
      LCD.clear();
    }
     // On odd ticks, print a message:
    else {
      LCD.print("Hello, world!");
    }

     // Update last with the current time and
     // increase tick
    last = millis();
    ++tick;
  }
}
```

EXERCISE 2

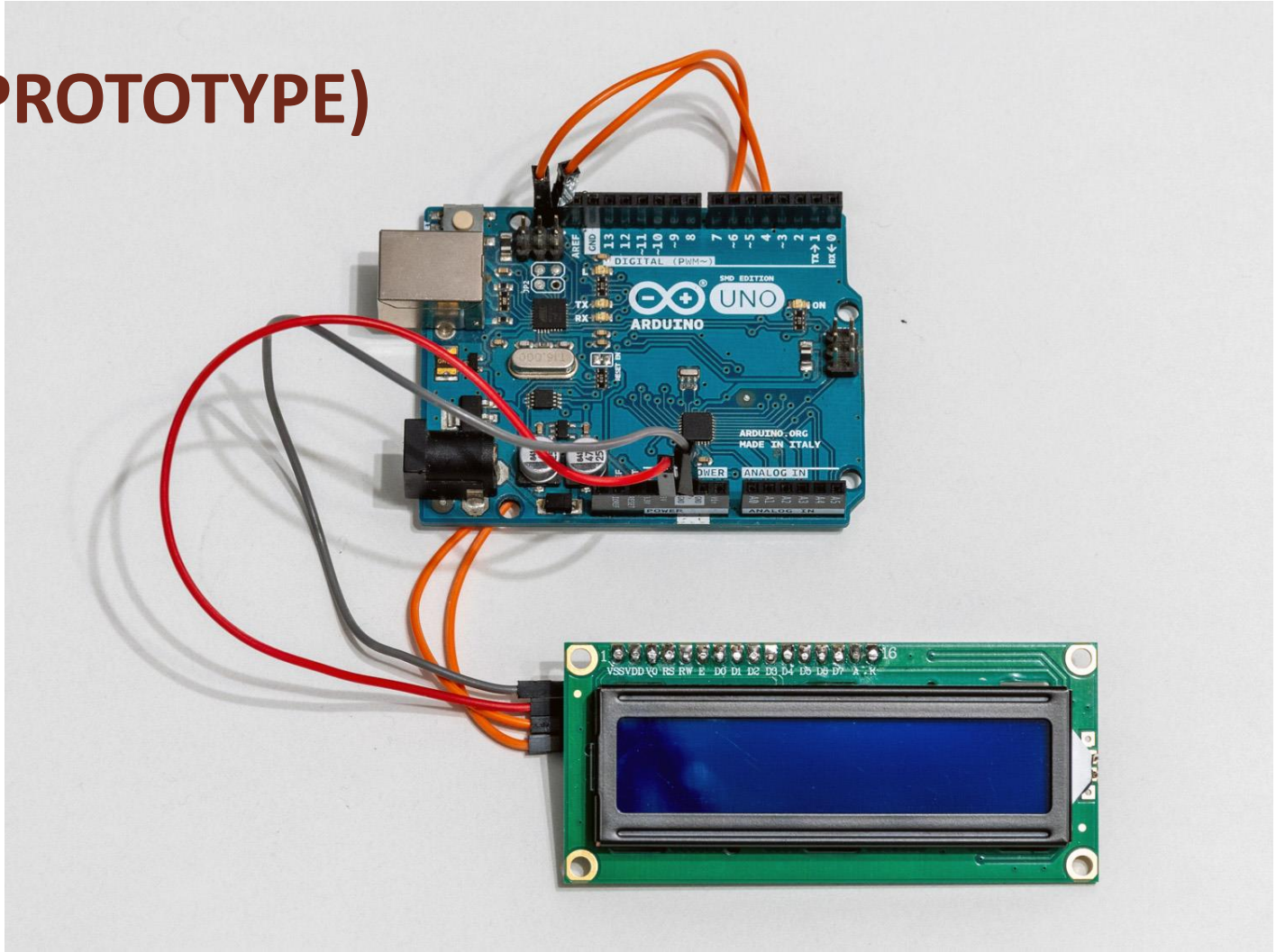
- Only slightly more advanced than the previous exercise. The user is required to enter any text on the Serial Monitor. Then, the program counts its' characters and writes the text's length on the 1st line of the LCD, plus the text itself on the 2nd line.
- This exercise will require you to work with text entered through the Serial Monitor (remember how to use it, right?) and text processing.



EXERCISE 2 (SCHEMATIC)



EXERCISE 2 (PROTOTYPE)



EXERCISE 2 (CODE – 1/3)

```
// Including the required libraries
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Creating the terminal symbol as a 5x8 HEX array
uint8_t terminalSymbol[8]  = { 0x00, 0x10, 0x08, 0x04, 0x08, 0x10, 0x07, 0x00 };

// Other variables
String  inputText;
size_t  inputLen;

// The LiquidCrystal object
LiquidCrystal_I2C LCD(0x3F, 16, 2);
```

EXERCISE 2 (CODE – 2/3)

```
void setup()
{
  // Start the UART port and wait for it to be ready
  Serial.begin(9600);
  while(!Serial)    ;

  // Initiate the LCD and light it up
  LCD.init();
  LCD.backlight();

  // Store the created character into the display's
  // memory
  LCD.createChar(0, terminalSymbol);

  // Home the LCD (position the cursor at 0,0) and print
  // some text
  LCD.home();
  LCD.print("L: ");

  // Position the cursor at 0, 1
  LCD.setCursor(0, 1);

  // Write byte '0'; the glyph that we created
  LCD.write(0);

} // END of setup()
```

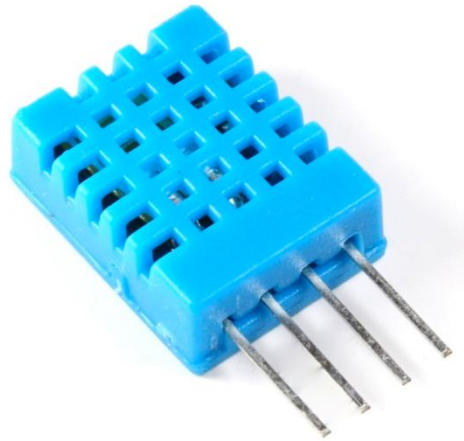
EXERCISE 2 (CODE – 3/3)

```
void loop()
{
  // If there's any incoming serial data…
  if (Serial.available())
  {
    // Read the data and size its' length
    inputText = Serial.readString();
    inputLen = inputText.length();

    // Clear the LCD and home it at the same time
    LCD.clear();

    LCD.print("L: ");

    // Position cursor at 2, 0 and print the length (num)
    // as a string
    LCD.setCursor(2, 0);
    LCD.print(String(inputLen));

    // Position cursor at 0, 1 and write byte 0 (the custom
    //glyph)
    LCD.setCursor(0, 1);
    LCD.write(0);

    // Finally, position the cursor ar 2, 1 and print the
    // input text
    LCD.setCursor(2, 1);
    LCD.print(inputText);

  } // END of IF
} // END of loop()
```

EXERCISE 3

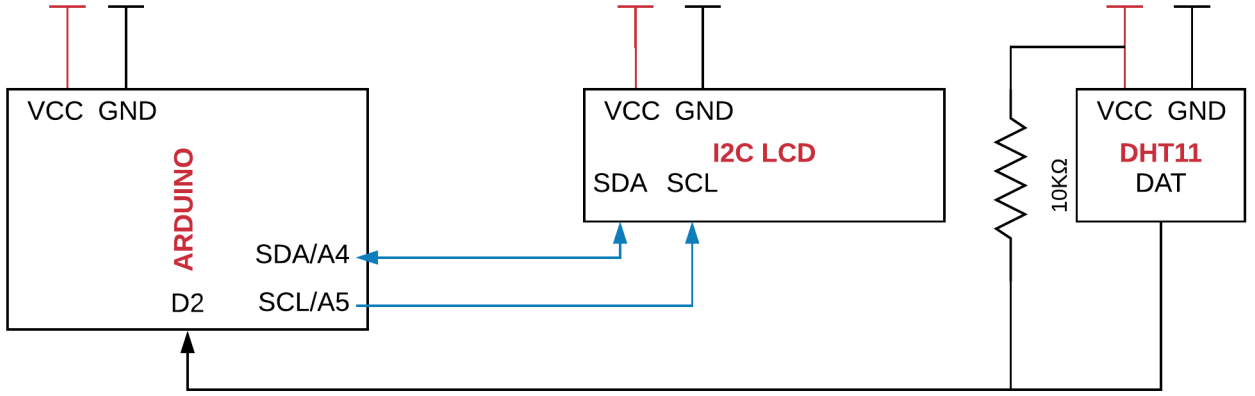
- Extending exercise 1, we now utilize data from a DHT11 combined temperature and humidity sensor. We are also making use of the extended ASCII table of our HD44780-based LCD display, in order to print the degrees symbol.

T	E	M	P	:		2	2	.	4	0		°	C		
H	U	M	D	:		5	0	.	1	0		%			

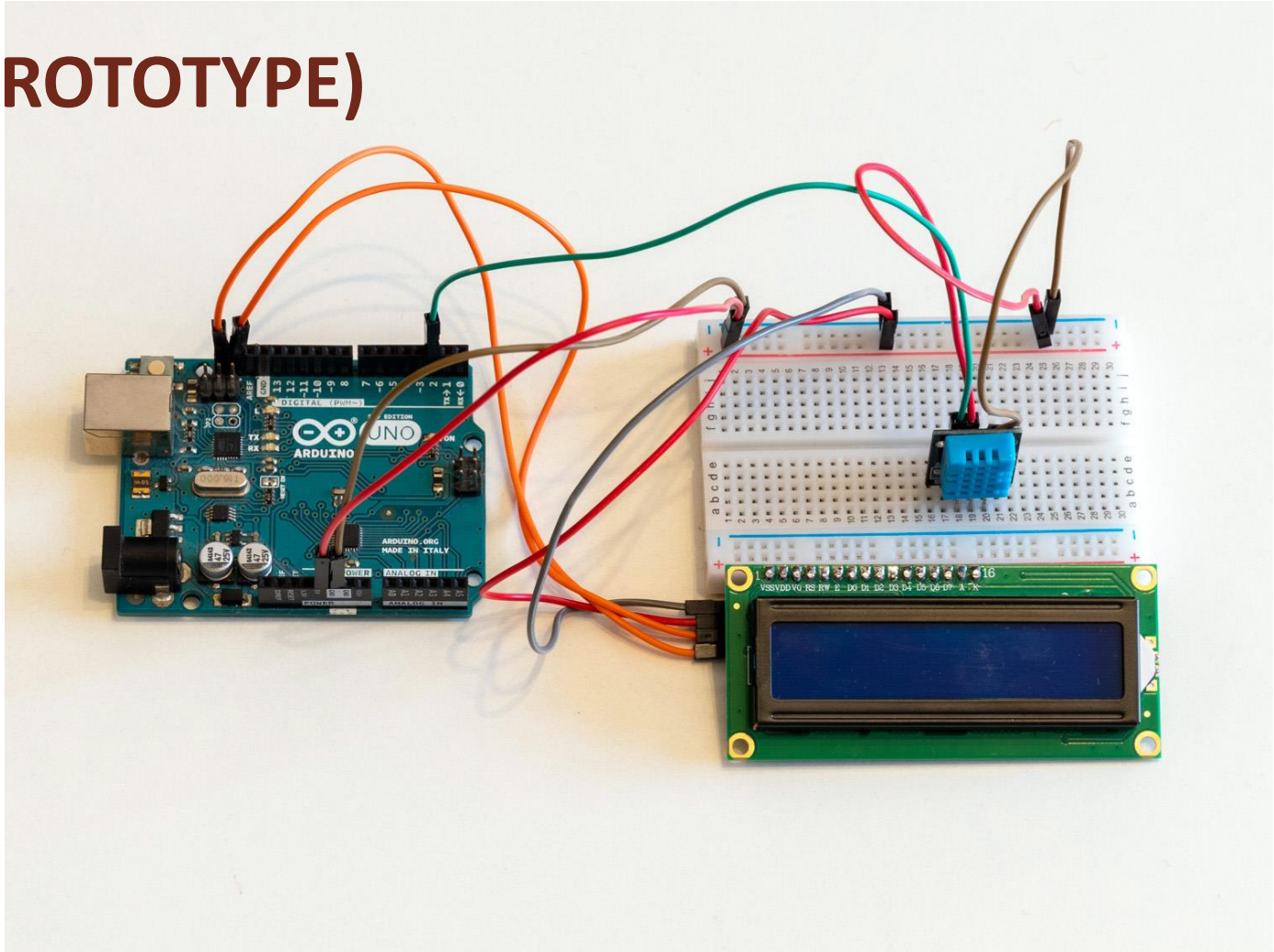


A DHT11 combined temperature and relative humidity sensor.

EXERCISE 3 (SCHEMATIC)



EXERCISE 3 (PROTOTYPE)



EXERCISE 3 (CODE – 1/3)

```
// Including the required libraries
#include “DHT.h” // INSTALL FIRST!
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Define the required data for our sensor e.g.
// type (DHT11) and pin on which it is connected (D2)
#define DHTPIN  2
#define DHTTYPE DHT11

// Init the sensor object with the above data
DHT sensor(DHTPIN, DHTTYPE);
```

```
// Float var for the temperature
float   tmp;

// Float var for the relative humidity
float   hum;

// The LiquidCrystal object
LiquidCrystal_I2C LCD(0x3F, 16, 2);
```

EXERCISE 3 (CODE – 2/3)

```
void setup()
{
  // Initialize the sensor
  sensor.begin();

  // Initiate the LCD and light it up
  LCD.init();
  LCD.backlight();

} // END of setup()
```

EXERCISE 3 (CODE – 3/3)

```
void loop()
{
  // The DHT11 sensor can only be read once every 2
  // seconds. Thus, we need to put a delay in the loop
  delay(2000);

  // Read values from the sensor and assign them
  tmp = sensor.readTemperature();
  hmd = sensor.readHumidity();

  // This is for error checking. Halt execution and
  // retry if we have erroneous values
  if ( isnan(tmp) || isnan(hmd) )
    return;

  LCD.home();
  LCD.print("TEMP: ");
  LCD.print(String(tmp) + " ");

  // Code 0xDF (ASCII HEX) is the degrees symbol!
  LCD.write(0xDF);
  LCD.print("C");

  LCD.setCursor(0, 1);
  LCD.print("HUMD: ");
  LCD.print(String(hum) + " ");
  LCD.print("%");

} // END of loop()
```

EXERCISE 3.1

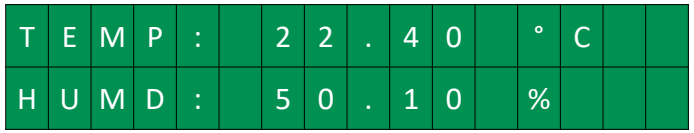
- Further extending exercise 3, we now deal with scrolling text, and simple output peripherals like an LED and a buzzer.

- If the DHT11 sensor reports values that are higher than our preset limits, an alarm will go off, producing an audible (buzzer) and visible (LED) warning. Moreover, the display will stop showing the data from the sensor, switching to a predefined warning text that scrolls indefinitely to the left

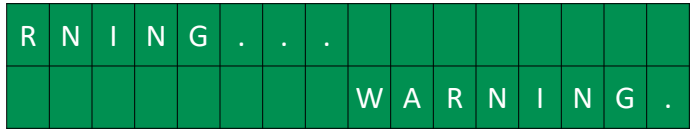
EXERCISE 3.1 (CONT.)

- This is a more advanced scenario to produce, as it will require multiple timekeeping operations and bigger control blocks.

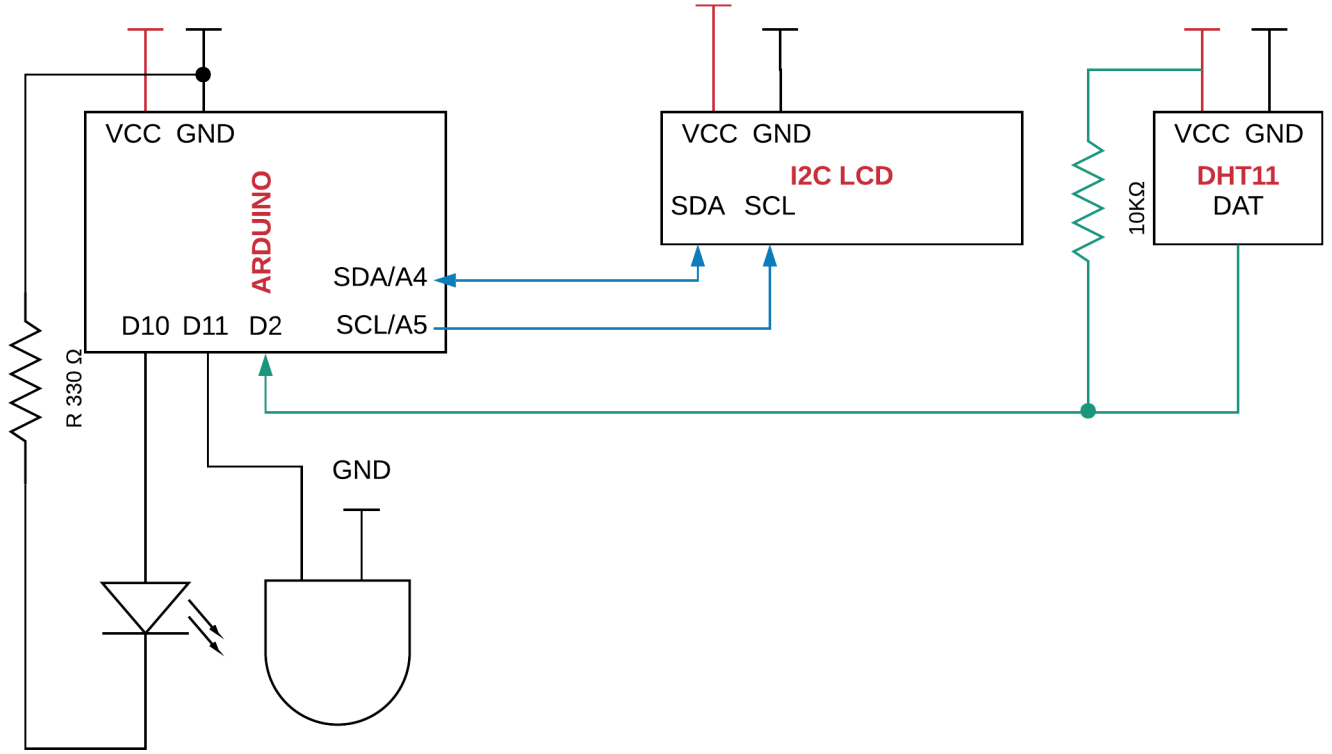
NORMAL STATE:



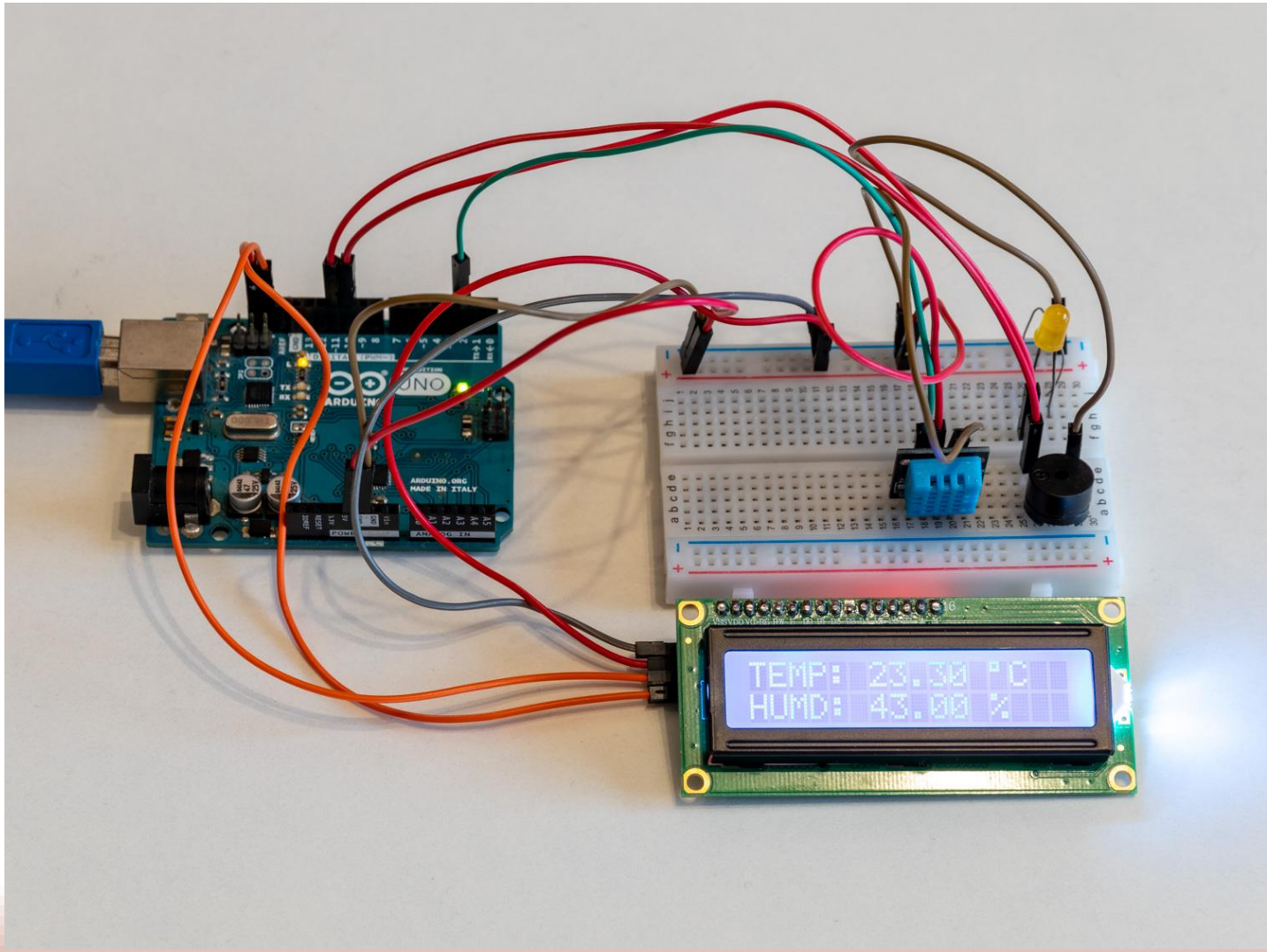
WARNING STATE:



EXERCISE 3.1 (SCHEMATIC)



EXERCISE 3.1 (PROTOTYPE)



EXERCISE 3.1 (CODE – 1/5)

```
// Including the required libraries
#include “DHT.h” // INSTALL FIRST!
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Define the required data for our sensor e.g.
// type (DHT11) and pin on which it is connected (D2)
#define DHTPIN  2
#define DHTTYPE DHT11

// Init the sensor object with the above data
DHT sensor(DHTPIN, DHTTYPE);
```

```
// First are the values read from the
// sensor, second are the set limits
float    tmp, tlim = 30;
float    hum, hlim = 80;

// All of these are for time-keeping
// operations!
unsigned long tick = 0;
unsigned long tock = 0;
unsigned long last = 0;
unsigned long beep = 0;

LiquidCrystal_I2C LCD(0x3F, 16, 2);
```

EXERCISE 3.1 (CODE – 2/5)

```
void setup()
{
  // Initialise the DHT sensor:
  sensor.begin();

  // Initialise the LCD:
  LCD.init();

  // Enable the backlight:
  LCD.backlight();

} // END of setup()
```

```
void loop()
{
  // Read values from the sensor every
  // 2 seconds (this sensor cannot be
  // read more often):
  if (millis() > last + 2000)
  {
    tmp = sensor.readTemperature();
    hum = sensor.readHumidity();

    // Error checking
    if (isnan(tmp) || isnan(hum))
      return;

     // Update the last execution time
     // with the current time:
    last = millis();
  }
```

EXERCISE 3.1 (CODE – 3/5)

```
// If either the temp, or the humidity is above
// the limits (which is considered al alarm), …
if ((tmp > tlim) || (hum > hlim))
  {
     // And tick is zero:
    if (!tick)
    {
       // Clear the LCD and print a warning
       // message
      LCD.clear();
      LCD.print("WARNING...");
      LCD.setCursor(0, 1);
      LCD.print("          WARNING...");
    }
```

```
else
{
   // Else (if tick is not 0)
   // move the text to the left:
  LCD.scrollDisplayLeft();
}

 // Increase tick, so that in the next loop,
 // the text moves instead on clearing again:
++tick;
```

EXERCISE 3.1 (CODE – 4/5)

```
// Check the beep timer, and fire pin 10 (LED)
// every 350ms (with the well established
// time-keeping trick) as well as beep the buzzer:
if (millis() > beep + 350)
{
  // Enable the LED pin and buzz every *other*
  // 350ms:
  if (tock % 2)
  {
    digitalWrite(10, HIGH);
    tone(11, 2500);
  }
  // Else, drive the pin LOW and stop the sound:
  else
  {
    digitalWrite(10, LOW);
    noTone(11);
  }
  // Now increase tock for the routine to fire
  // (on-off) every other 350ms:
  ++tock;

  // Lastly, keep track of the timestamp in order
  // for the alternation to work:
  beep = millis();
 }
}
```

EXERCISE 3.1 (CODE – 5/5)

```
// If there is NO warning state, …
else
  {
     // Disable the warning alternation functions:
    tick = 0;
    tock = 0;
     // Drive the LED pin LOW and disabled the
     // buzzer:
    digitalWrite(10, LOW);
    noTone(11);
     // Clear the LCD and print out the
     // sensor values:
     LCD.clear();
    LCD.print("TEMP: ");
    LCD.print(String(tmp) + " ");
    LCD.write(0xDF);
    LCD.print("C");

    LCD.setCursor(0, 1);
    LCD.print("HUMD: ");
    LCD.print(String(hum) + " ");
    LCD.print("%");

  } // END of else

  delay(350);

} // END of loop()
```

EXERCISE 4

- Combining what we've learnt from the previous exercises, we are going to replicate a pharmacy display. Most pharmacies have a display that not only shows the "Hygeia basin and snake", but also the date, time, and sometimes even the temperature
- On the 1st line of the display, we are going to display the Hygeia basin and snake, which is a custom glyph, alongside a predefined text like "Welcome!“
- On the 2nd line, however, we are going to switch from displaying the current date and time, to showing the temperature, as read from the DHT11.

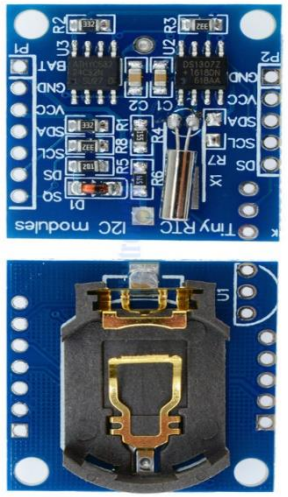
The "Hygeia basin and snake", symbol of the ancient Greek goddess of health, Hygeia.

		P	H	A	R	M	A	C	Y		O	P	E	N	
1	9	/	0	4	/	2	0	2	0		2	0	:	2	3

		P	H	A	R	M	A	C	Y		O	P	E	N	
T	E	M	P	:		2	2	.	4	0		°	C		

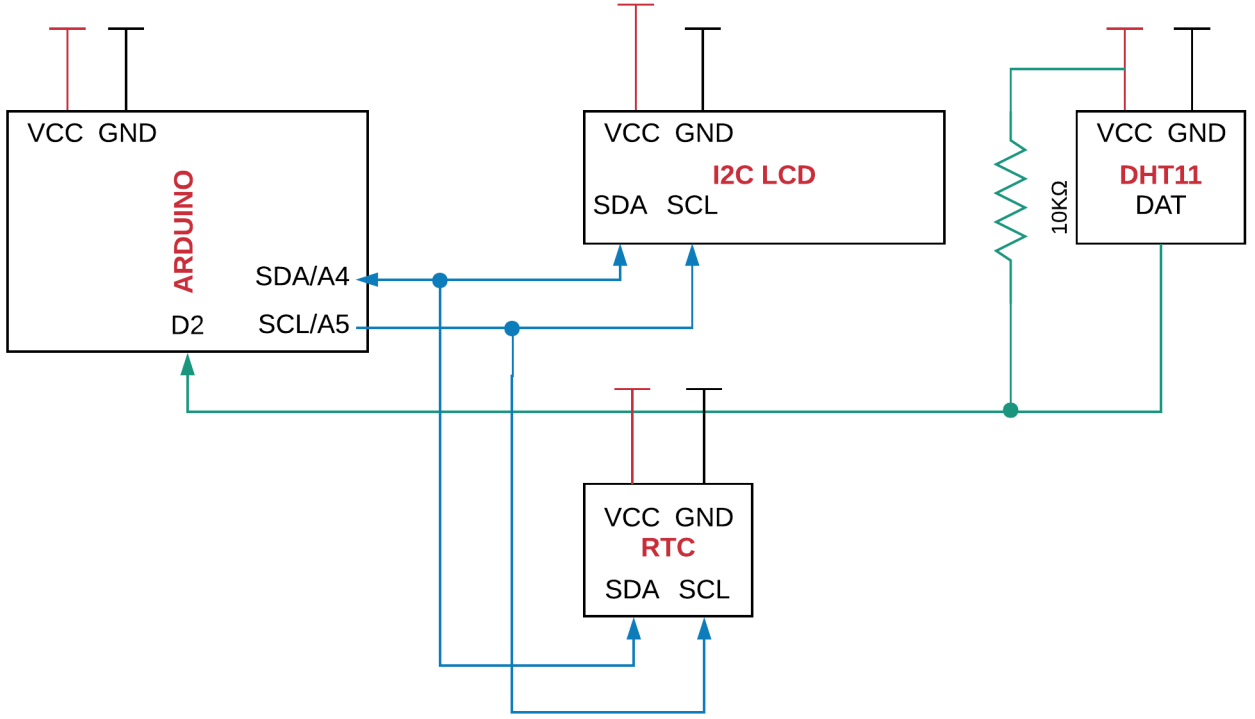
EXERCISE 4 (CONT.)

- You might wonder "How we are going to retrieve the current date and time?"
- We will need to use yet another I2C peripheral device that is called "an RTC" (real-time clock) for this purpose. Being an I2C device as well, it can reside on the same bus with the display without any conflict

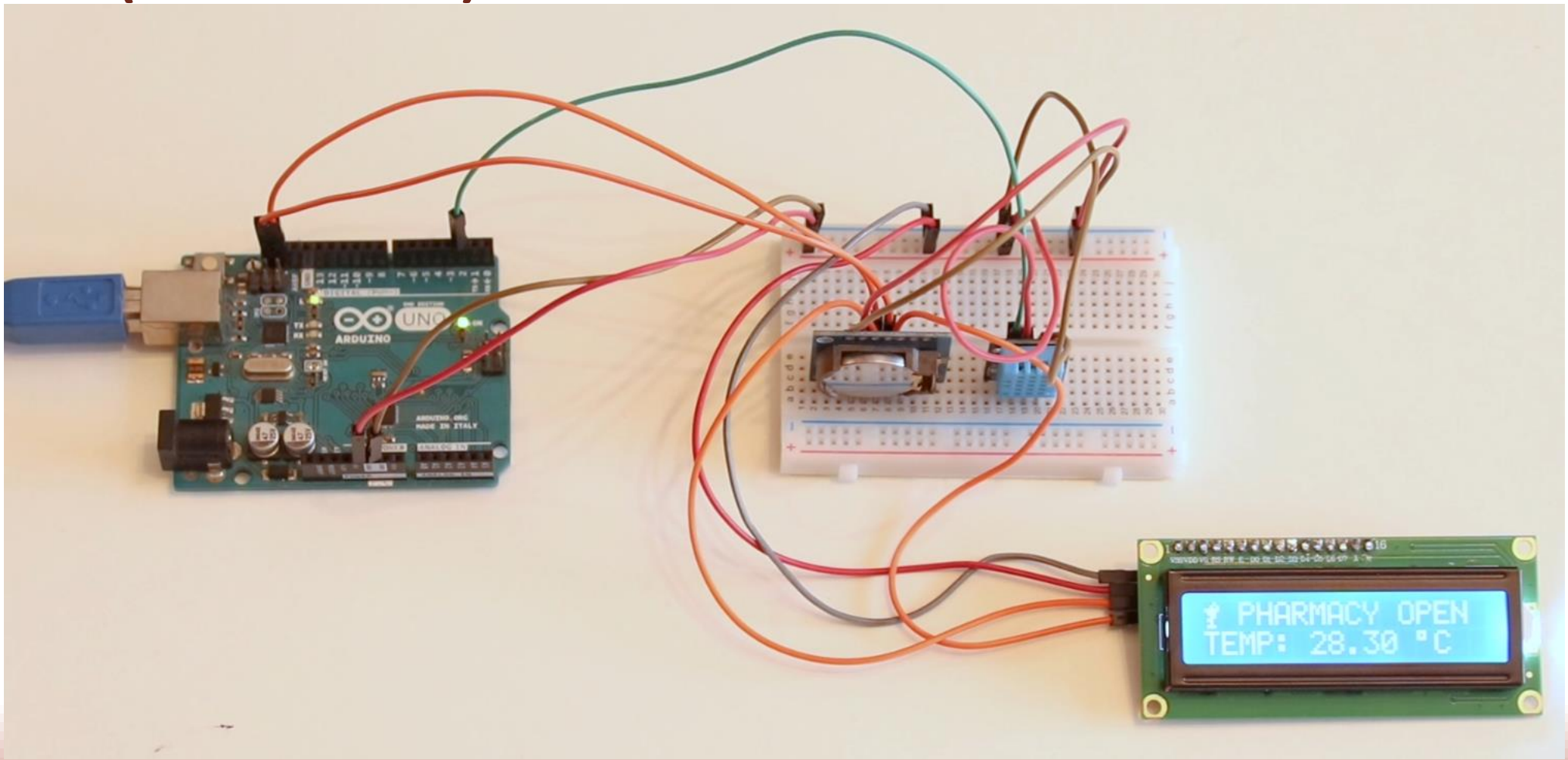


A common RTC module, based on the DS1307 IC.

EXERCISE 4 (SCHEMATIC)



EXERCISE 4 (PROTOTYPE)



EXERCISE 4 (CODE – 1/3)

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

// Library for the Real-Time Clock (RTC) module:
#include "RTClib.h“ // INSTALL FIRST!
#include "DHT.h"

#define DHTPIN 2
#define DHTTYPE DHT11

// Construct a DHT type object:
DHT sensor(DHTPIN, DHTTYPE);
```

```
// Holds the current temperature:
float    tmp;
// The DateTime object (named now) is a special
// structure that incorporates date and time:
DateTime now;

unsigned long tick = 0;

LiquidCrystal_I2C LCD(0x3F, 16, 2);

// Construct an RTC object:
RTC_DS1307 RTC;

// These will define the symbol presented as a pharmacy
// snake:
uint8_t pharmacySnake[8]  = { 0x02, 0x05, 0x1F, 0x0E, 0x06,
0x0C, 0x04, 0x0E };
```

EXERCISE 4 (CODE – 2/3)

```
void setup()
{
  // Init the DHT sensor and wait for 2s (this is
  // required by the sensor) before reading:
  sensor.begin();
  delay(2000)   ;

  // If the RTC cannot initialise halt execution;
  // while(1) is an infinite loop:
  if(!RTC.begin())
    while(1);

  // Check if the RTC already has a date in
  // its registers:
  if (! RTC.isrunning())
     // If not, set it, using compiler macros (current
     // system time):
    RTC.adjust(DateTime(F(__DATE__), F(__TIME__)));


  LCD.init();
  LCD.backlight();

  // This will store the snake symbol in the 1st
  // “spot” in the LCD’s memory:
  LCD.createChar(0, pharmacySnake);

  LCD.home();
  LCD.write(0);
  LCD.print(" PHARMACY OPEN");

}
```

EXERCISE 4 (CODE – 3/3)

```
void loop()
{

  now = RTC.now(); // Get the current time
  tmp = sensor.readTemperature(); // And temp
  // This will clear the LCD:
  LCD.setCursor(0, 1);
  LCD.print("                ");

  // With our established trick, display
  // every other time, either the date and time:
  if ( tick % 2 != 0)
  {
    LCD.setCursor(0, 1);
    LCD.print(String(now.day(),   DEC) + "/");
    LCD.print(String(now.month(), DEC) + "/");
    LCD.print(String(now.year(),  DEC) + " ");
    LCD.print(String(now.hour(),  DEC) + ":");
    LCD.print(String(now.minute(),DEC));
  }

  // Or the temperature:
  else
  {
    LCD.setCursor(0, 1);
    LCD.print("TEMP: " + String(tmp) + " ");
    LCD.write(0xDF);
    LCD.print("C");
  }


  ++tick; // Increase tick for the trick to work
  delay(3000); // Do this every 3s

} // END of loop()
```

CONGRATULATIONS

You have completed SYS-STEM Module 6