LAB 1 - Seeeduino Nano / Seeeduino XIAO Essentials

Ana Lopes (98587)[1] and Mariana Mourão (98473)[2]

[1] *Instituto Superior Técnico,*

Integrated Master's in Biomedical Engineering,

ana.rita.santos.lopes@tecnico.ulisboa.pt

[2] *Instituto Superior Técnico,*

Integrated Master's in Biomedical Engineering,

mariana.mourao@tecnico.ulisboa.pt

Exercises:

Note: For each code sketch, the variables data types were chosen in a way to minimize the required memory resources, by concluding which is the maximal representation space that variables need. Furthermore, note that each code sketch is displayed as a print screen of the Arduino IDE.

1. An alternative and optimized formulation of the 01.BASICS/BLINK sketch that only uses a single call to the digitalWrite() and delay() functions (deletes code duplication) is displayed in Figure 1. The corresponding circuit montage is illustrated in Figure 2.

```
const byte LED = 13; // sets the pin number as a constant that won't change, being
                     // of type byte for optimizing memory resources (8 bits, rather
                     // than 16 bits of the int data type)

bool LEDstate = LOW; //initializes the LED state, being of type bool (1 bit) since
                     // there are only two possible states

// the setup routine runs once when you press reset or power the board
void setup(){
  pinMode(LED, OUTPUT); // set the digital pin as output
}

// the loop routine runs over and over again forever:
void loop(){
  digitalWrite(LED, LEDstate); // attributes the LED state
  delay(1000); // wait for a second
  LEDstate = !LEDstate; // invert the LED state
}
```

Figure 1 – Code sketch for exercise 1.

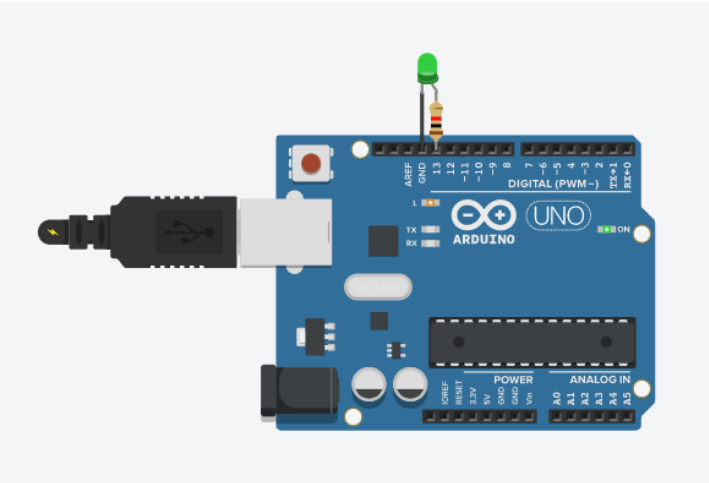

Figure 2 – Circuit montage for exercise 1, making the LED blink with a given frequency (1 second in this case). A 220k ohm resistor is connected to the pin 13 (digital entry) of the board, connecting the LED's anode to the resistor. The LED's cathode is connected to the board GND.

2. The possible drawbacks of using the delay function result from the mandatory waiting time for executing another action (reading of sensors, mathematical calculations and/or pin manipulation), as it pauses the program for the amount of time (in milliseconds) specified. An alternative formulation of the 01.BASICS/BLINK sketch that produces the same result without using the delay() function could be achieved with the millis() function, a command that returns the number of milliseconds since the board started running its current sketch. In each iteration of the main loop, the current value of millis() is compared with the start time (set to 0), determining whether or not the desired blink time has passed. In case it has, it inverts the LED state and sets the start time to the current time value. In this way, the LED blinks continuously while the sketch execution never lags on a single instruction, allowing other code to run at the same time without being interrupted by the LED code. In Figure 3 is displayed the code sketch.

```
const byte LED = 13; // sets the pin number as a constant that won't change, being
                     // of type byte for optimizing memory resources (8 bits, rather
                     // than 16 bits of the int data type)
bool Ledstate = LOW; //initializes the LED state, being of type bool (1 bit) since
                     // there are only two possible states

// Generally, you should use "unsigned long" for variables that hold time
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0;  // initializes the last time (in milliseconds) the LED state
                                   // was updated. Since time values can quickly become too large
                                   // for an int to store, the "unsigned long" data type was used

const int interval = 1000; // time interval at which to blink (in milliseconds), defined as a
                           // constant integer value (won't change)

unsigned long currentMillis; // creates the variable for controlling time

// the setup routine runs once when you press reset or power the board
void setup() {
  pinMode(LED, OUTPUT); // set the digital pin as output
}

// the loop routine runs over and over again forever:
void loop() {
  currentMillis = millis(); //get the current time (number of milliseconds since the program started)

  // check if the desired blink time has passed, by comparing if the current time and last time
  // the LED blinked is bigger than desired blink time
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis; // saves the last time (in milliseconds) the LED state was updated
    Ledstate = !Ledstate; // invert the LED state

    digitalWrite(LED, Ledstate); // attributes the LED state
  }
}
```

Figure 3 – Code sketch for exercise 2.

3. On the 01.BASICS/DigitalReadSerial sketch, a variable of type integer (int, 16 bits for its representation) is taken as input to the Serial.println() command, intending for its value to be printed. This command prints any data type to the serial port as human-readable ASCII, implying its conversion to the character data type (char, 8 bits for its representation), thus, in practice, it would be transferring 8 bits. Although, besides these characters being transferred, it is also added a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n'), each one requiring 8 bits for their representation, resulting in 24 bits being transferred. Note that this additional data will make possible for the Serial.println() command to always transmit 16 bits, even though any input data is given.

This serial communication follows a protocol, for the purpose of the receiver to decipher the data being sent by the system. For this, the data bits transmitted (char data type) are packed into frames, created by appending synchronization and parity bits to the data. The data bits refer to the effective data (5-9 bits), where the most commonly used 8-bit byte. The synchronization bits represent the start (1 bit) and stop bits (1-2 bits), that mark the beginning and ending of a packet, respectively. Finally, the parity (odd or even) bit is

optional (0-1 bit), being set by the evenness of the sum of the data bits. These protocols are configured by the Serial.begin() command, having as default the SERIAL_8N1 protocol (8 data bits, no parity, 1 stop bit and 1 start bit = 10 bits). For the 01.BASICS/DigitalReadSerial sketch, the default protocol is applied, being pushed 10 bits to the serial port for each character, plus the 20 bits (\r and \n) implied in the Serial.println() command.

4. In this exercise, it was intended to determine the average time that the Serial.println() instruction (01.BASICS/ReadAnalogVoltage sketch) takes to execute on the microcontroller, evaluating (debugging) the code performance regarding data transmission. For this, the millis() function was used for accessing, in each iteration, the time (in milliseconds) just before and after executing the Serial.println() instruction, computing the difference between these values for obtaining an estimate of its execution time. In order to obtain estimates statistically more significant, an average across iterations was performed on the microcontroller, streaming the data to the Serial Plotter (figure 6). In Figure 4 is displayed the code sketch, with the corresponding circuit montage illustrated in figure 5.

```
// NOTE: Since time values can quickly become too large for an int to store, the
// "unsigned long" data type was used for storing time variables
unsigned long start_time; // creates the variable for starting the count, in each iteration, of
                          // the execution time of the Serial.println command, by accessing
                          // through the millis function the number of milliseconds since tne program started
unsigned long end_time; // creates the variable for ending the count, in each iteration, of the execution
                        // time of the Serial.println command, by accessing through the millis function the
                        // number of milliseconds since tne program stated
unsigned long delta; // creates the variable that stores the execution time on a given iteration
unsigned long sum_delta = 0; // initializes the variable that stores the sum of deltas across iterations
int num = 0; // initializes the number of iterations

// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  pinMode(1, OUTPUT); // set the digital pin as output
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
  int sensorValue = analogRead(A0);
  // Convert the analog reading (which goes from 0 - 1023) to a voltage (0 - 5V):
  float voltage = sensorValue * (5.0 / 1023.0);


  start_time = millis(); // accesses the number of milliseconds since the program started, initiating the count
                         // of execution time of the current iteration
  Serial.println(voltage); // print out the value you read
  end_time = millis(); // accesses the number of milliseconds since the program started, ending the count of
                       // execution time of the current iteration

  delta = end_time - start_time; // computes the execution time

  num += 1; // increments the number of iterations
  sum_delta += delta; // uptades the sum of deltas across iterations

 Serial.print(sum_delta/ (float) num); Serial.print("  ");
 // Casting of the num variable to the float type for computing the average of execution times across iterations.
 // If it was of type int, when performing division the result will also be an int, with any remainder (i.e., digits
 // after the decimal point) discarded.
}
```

Figure 4 – Code sketch for exercise 4.

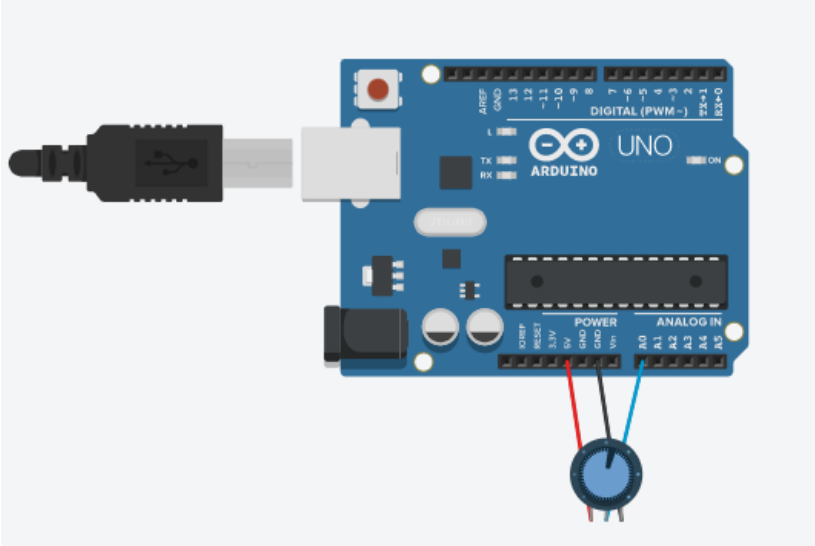

Figure 5 – Circuit montage for exercise 4. A 10k ohm potentiometer is used, attaching its center pin to pin A0 (analog entry) on the board, and its outside pins to the +5V and ground pins on the board.

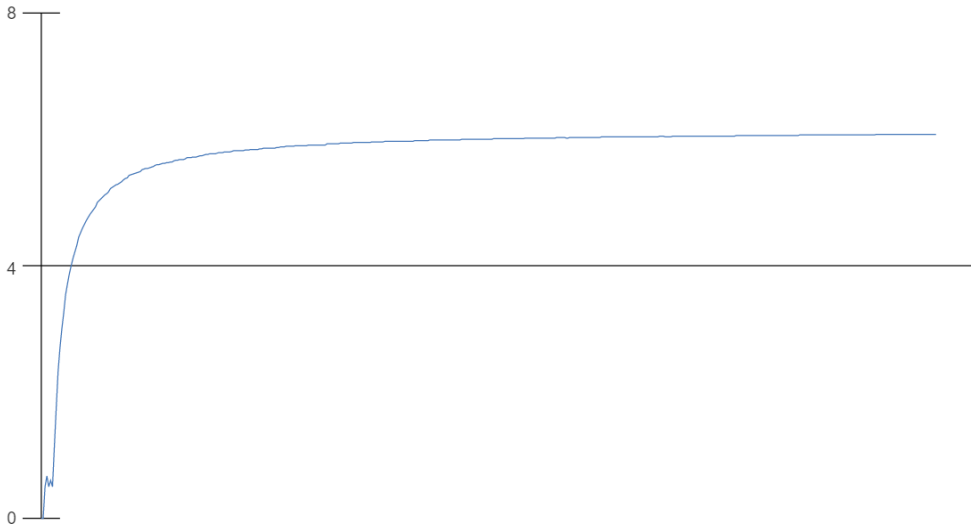

Figure 6 – Serial Plotter, streaming the updated average of the execution times across iterations, associated to the Serial.println(voltage) instruction. The x axis represents the number of iterations, and the y axis the updated average.

By analysing figure 6, the time average estimates progressively tends to a given value across iterations, starting from a value of approximately 0 milliseconds (if using the micros() command it wouldn't be 0 milliseconds, but of the order of 10^2 microseconds) and reaching the value of 6.08 milliseconds (accessed on the serial monitor).

In order to evaluate if the observed follows what would be expected, a theoretical estimate is computed taking into account the conclusions derived on exercise 3, regarding the established protocol for serial communication. In the 01.BASICS/ReadAnalogVoltage sketch, the serial communications is configured by the Serial.begin() command, stablishing

9600 bits of data per second (baud rate) transferred between the computer and the Arduino, adopting the default serial protocol (SERIAL_8N1, with 10 bits being pushed to the serial port per character). The baud rate specified gives a time interval of 0.000104 seconds (0.104 milliseconds) per bit of data transferred. On the other hand, the analog values to be printed on the serial port range from 0.00 to 5.00 (float data type), which are converted to the char data type, needing 4 different characters for their representation. Additional to these 4 characters, the '\r' and '\n' characters are also appended, which makes, in practice, the Serial.println() command print 6 characters. This way, 60 bits (6 characters * 10 bits/character = 60 bits) are transferred each time the instruction Serial.println(voltage) is performed, having an execution time of 60 bits × 1.04 milliseconds/bit = 6.24 milliseconds (theoretical estimate).

Comparing the theoretical and experimental estimates, it can be concluded that the Serial.println() instruction has an execution time predicted by the reasoning applied for computing the theoretical estimate, which considers the baud rate stablished. Although, note that this does not applies to the microcontrollers whose serial communication protocol stablishes a fixed baud rate, with the Serial.begin() command not affecting in any way the baud rate at which the data is transmitted. This is the case of the Seeeduino XIAO, the smallest Arduino compatible board in Seeeduino family, which uses a different serial communication chip set (USB CDC), transmitting the data at the maximum speed. Thus, for seeeduino XIAO, the experimental estimate would be lower than the theoretical estimate shown above, since the data is transmitted at a higher speed than was stablished by the Serial.begin() command. As for the seeeduino NANO, the serial port speed is configurable by the Serial.begin() command, with the experimental estimates being closer to the theoretical estimate.

Reflecting on the nature of the execution time estimation, it translates the elapsed time on the embedded system, not being affected by the computer transmission. Since the embedded system is a limited resources device, characterizing and taking into account the execution times of different instructions when developing embedded applications is indeed crucial, since although they seem to be sequentially executed, they actually have a time cost associated.

5. For this exercise, a bidirectional communication to the system was stablished, by sending specific characters for the system to respond, controlling the streaming of the analog values through the serial port. The selected characters were 'S' for the system to stream the data ('ON' state), and 'E' for the system to stop streaming the data ('OFF' state). When the system's state is 'ON', the variation of the analog values, controlled by the potentiometer, are graphically represented on the Serial Plotter graph (figure 9). In Figure 7 is displayed the code sketch, with the corresponding circuit montage illustrated in figure 8.

```
int sensorPin = A0; // select the input pin for the potentiometer
const byte LED = 5; // sets the pin number as a constant that won't change, being
                    // of type byte for optimizing memory resources (8 bits, rather
                    // than 16 bits of the int data type)
int sensorValue = 0;  // initializes the analog value coming from the sensor, and since it is
                      // converted to a digital number from 0-1023, the data type int is the
                      // more suitable
char CH; // creates the variable for storing the char coming from the serial port
bool system_state = 0; // initializes the state of the system (0 being off and 1 being on)
bool LED_state = LOW; // initializes the LED state
unsigned long previousMillis = 0; // initializes the last time (in milliseconds) the LED state
                                  // was updated. Since time values can quickly become too large
                                  // for an int to store, the "unsigned long" data type was used
unsigned long currentMillis; // creates the variable for controlling time

// the setup routine runs once when you press reset or power the board
void setup(){
   Serial.begin(9600); // open the serial port and sets the data rate to 9600 bps
   pinMode(LED, OUTPUT); // set the digital pin as output
}

// the loop routine runs over and over again forever:
void loop(){

  // reply only when data is received
  if (Serial.available()>0)
  {
    CH = Serial.read(); // read the incoming byte

    // set the system's state for the characters defined to control the data streaming. If other
    // character is received, the system's state won't be modified
    switch(CH)
    {
      case 'S': // character for starting streaming
      system_state = 1; // system state ON - LED blinking and data streaming
      break;

      case 'E': // character for ending streaming
      system_state = 0; // system state OFF - LED set to LOW voltage and data not being streamed
      break;
    }
  }

  if (system_state == 1)  // streams data and checks if the LED state is to be updated
  {
    sensorValue = analogRead(sensorPin); // reads the value from the specific analog pin

    currentMillis = millis(); //get the current time (number of milliseconds since the program started)

    // check if the blink time set by the analog sensor has passed, by comparing if the current time
    // and last time the LED blinked
    if (currentMillis - previousMillis >= sensorValue)
    {
      previousMillis = currentMillis; // saves the last time (in milliseconds) the LED state was updated

      LED_state = !LED_state; // invert the LED state

      digitalWrite(LED, LED_state); // attributes the LED state
    }

    Serial.println(sensorValue); // prints the analog reading
    delay(10); // Delay a little bit to improve simulation performance
  }

  else if (system_state == 0) // stops data streaming and sets the LED state to LOW voltage
  {
    digitalWrite(LED, LOW); // attributes LOW voltage to the LED
  }
}
```

Figure 7 – Code sketch for exercise 5.

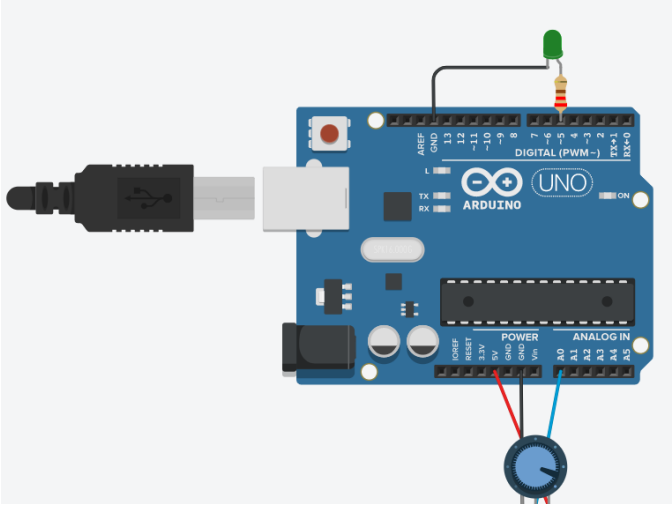

Figure 8 – Circuit montage for exercise 5. A10k ohm potentiometer is used, attaching its center pin to pin A0 (analog entry) on the board, and its outside pins to the +5V and ground pins on the board. A 220k ohm resistor is connected to the pin 5 of the board, connecting the LED's anode to the resistor. The LED's cathode is connected to the board GND.

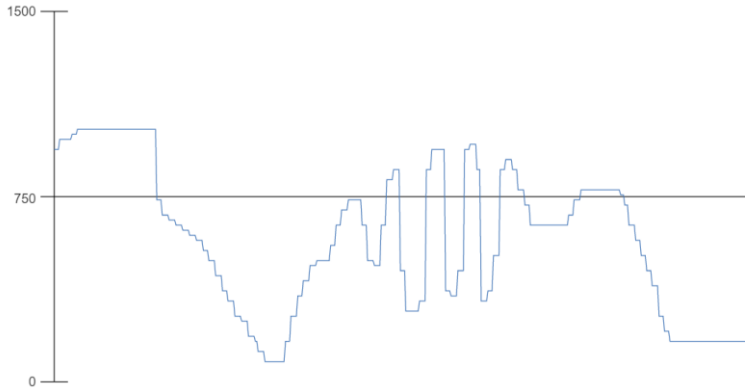

Figure 9 – Serial Plotter, streaming the variation of the analog values, controlled by the potentiometer. The x axis represents the number of iterations, and the y axis corresponds to the analog reading (range from 0-1023).

Regarding the Serial Plotter graph, the x axis represents the number of iterations, and the y axis corresponds to the analog reading (goes from 0 – 1023, since the input voltage range, 0 to 5 volts, is converted to a digital value by the analog-to-digital converter circuit). The recorded values translate the manual variation of the potentiometer shaft, which changes the amount of resistence on either side of the wiper connected to the center pin of the potentiometer, reflecting in a different analog input that will dynamically control the oscillation frequency of the LED. For the values closer to 0 (potentiometer's shaft turned all the way in one direction), the LED appears to be always ON, since the LED states transition immediately. Whereas, for progressively higher values (potentiometer's shaft turned all the way in the opposite direction) the blink time of the LED is longer, being perceptible the transition between LED states. The signal plateaus correspond to the time instants that the potentiometer shaft did not change position.