

Chapter 2

The Basic Functions

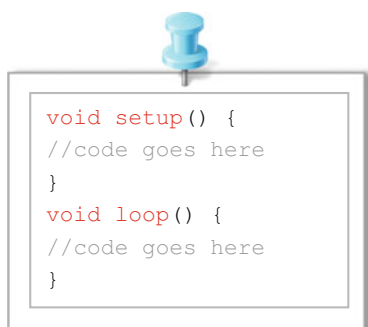
2.1 Overview

The code you learn to write for your Arduino is very similar to the code you write in any other computer language. This implies that all the basic concepts remain the same and it is simply a matter of learning a new dialect. In the case of Arduino, the language is based on the C/C++ and can even be extended through C++ libraries. The IDE enables you to write a computer program, which is a set of step-by-step instructions that you then upload to the Arduino. Your Arduino will then carry out those instructions and interact with whatever you have connected to it. The Arduino includes many basic embedded functions, such as the functions for reading and writing to digital and analog input and output pins, interrupt functions, mathematical functions, and serial communication functions. Arduino functions are a convenient way to write code such as those for device drivers or commonly used utility functions. Furthermore, Arduino also consists of many built-in-examples. You just need to click on the toolbar menu: **File** → **Examples** to access them. These simple programs demonstrate all basic the Arduino commands. They span from a Sketch Bare Minimum, Digital, and Analog IO, to the use of Sensors and Displays.

For more information on the Arduino language, see the Language Reference section of the Arduino web site, <http://arduino.cc/en/Reference/HomePage>. All Arduino instructions are online.

2.2 Structure

The basic function of the Arduino programming language is fairly simple and runs in at least two parts. These two required parts, or functions, enclose blocks of statements.



setup(): A function present in every Arduino sketch. Run once before the `loop()` function. The `setup()` function should follow the declaration of any variables at the very beginning of the program. It is the first function to run in the program, is run only once, and is used to set `pinMode` or initialize serial communication.


loop(): A function present in every single Arduino sketch. This code happens over and over again—reading inputs, triggering outputs, etc. The `loop()` is where (almost) everything happens and where the bulk of the work is performed.

2.3 Digital I/O Functions

Digital I/O will allow us to read the state of an input pin as well as produce a logical high or low at an output pin. If every potential external connection between a microcontroller and the outside world had a dedicated wire, the pin count for controller packages would be high. The ATmega 328P in the Romeo board has four 8-bit ports plus connections for power, ground and the like, yet it only has 28 physical pins. In general, each bit of a port can be programmed independently; some for input, some for output, or all of them for the same purpose.

1. `pinMode(pin, mode)`

Before we use a port, we need to inform the controller about how it should operate. In the Arduino system, this is usually done via a call to the library function `pinMode()`. Here is a description of the function from online references:



```
pinMode(pin,mode)
```

Parameters

pin: the number of the pin whose mode you wish to set

mode: INPUT, OUTPUT, or INPUT_PULLUP

Returns

None

It should be noted that the pin could be a number or variable with a value ranging from 0 to 13 or A0 to A5 (when using the Analog Input pins for digital I/O) corresponding to the pin number printed on the interface board. Furthermore, Digital pins default as input, so you really only need to set them to OUTPUT in `pinMode()`.

2. **digitalWrite(pin, value)**

Once a pin is established as an OUTPUT, it is then possible to turn that pin on or off using the `digitalWrite()` function. Its syntax is as follows:



```
digitalWrite(pin,value)
```

Parameters

Pin: the number of the pin you want to write


value: HIGH or LOW

Returns

None

3. `digitalRead(pin)`

With a digital pin configured as an INPUT, we can read the state of that pin using the `digitalRead()` function. Its syntax is as follows:



```
digitalRead(pin)
Parameters
    pin: the number of the pin you want to read (int)
Returns
    HIGH or LOW
```

The pin can be specified as either a variable or constant (0–13) and the result is either HIGH or LOW.

4. Example

The following example reads a pushbutton connected to a digital input and turns on an LED connected to a digital output when the button is pressed. The circuit is shown in Fig. 2.1

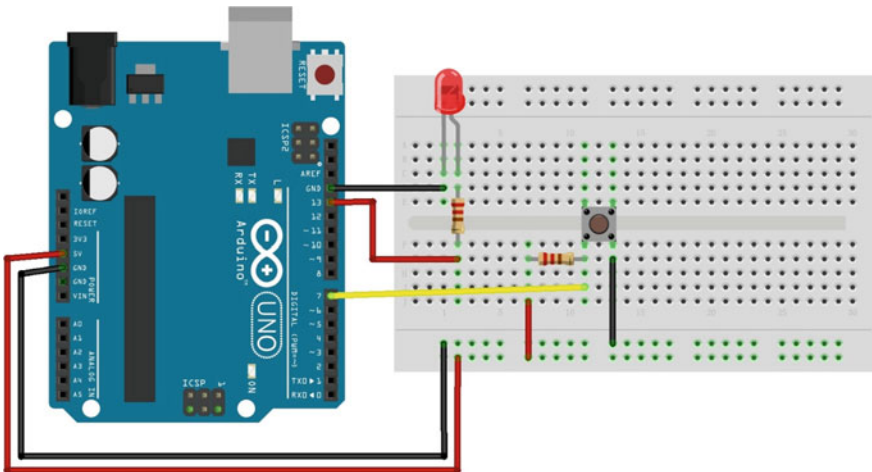


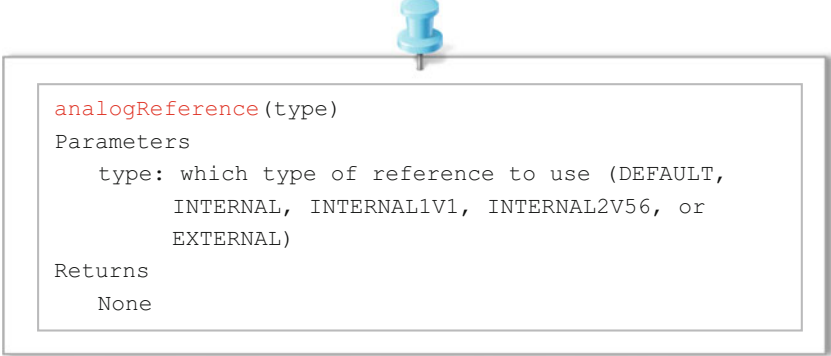
Fig. 2.1 Circuit layout for digital signal read and write

```
1  int led = 13; // connect LED to pin 13
2  int pin = 7;  // connect pushbutton to pin 7
3  int value = 0; // variable to store the read value
4
5  void setup() {
6      pinMode(led, OUTPUT); // set pin 13 as output
7      pinMode(pin, INPUT);  // set pin 7 as input
8  }
9  void loop() {
10     value = digitalRead(pin); // set value equal to the pin 7 input
11     digitalWrite(led, value); // set LED to the pushbutton value
12 }
13
```

2.4 Analog I/O Functions

1. `analogReference(type)`

The Arduino interface board, however, has a convenient pin called AREF located near digital pin 13 along with a function called `analogReference()` to provide the Arduino's ADC a reference voltage other than +5 V. This function will effectively increase the resolution available to analog inputs that operate at some other range of lower voltages below +5 V. The syntax for this function is as follows.



```
analogReference(type)
Parameters
    type: which type of reference to use (DEFAULT,
        INTERNAL, INTERNAL1V1, INTERNAL2V56, or
        EXTERNAL)
Returns
    None
```

DEFAULT: the default analog reference of 5 volts (on 5 V Arduino boards) or 3.3 volts (on 3.3 V Arduino boards)

INTERNAL: a built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328 and 2.56 volts on the ATmega8 (not available on the Arduino Mega)

INTERNAL1V1: a built-in 1.1 V reference (Arduino Mega only)

INTERNAL2V56: a built-in 2.56 V reference (Arduino Mega only)

EXTERNAL: the voltage applied to the AREF pin (0–5 V only) is used as the reference.

The function is only called once in a sketch, but must be declared before `analogRead()` is used for the first time, making it a suitable candidate for placing in the `setup()` function. The type specified relates to the kind of reference voltage that we want to use.

2. `analogRead(pin)`

Reads the value from a specified analog pin with a 10-bit resolution. This function works with the above analogy only for pins (0–5). The `analogRead()` command will return a number including or between 0 and 1023.



```
analogRead(pin)
```

Parameters

pin: the number of the analog input pin to read from
(0–5)


Returns

int(0 to 1023)

It takes about 100 μ s (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second. Furthermore, analogy pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

3. `analogWrite(pin,value)`

The Arduino also has the capability to output a Digital signal that acts as an Analog signal, this signal is called pulse width modulation (PWM). Digital Pins # 3, # 5, # 6, # 9, # 10, and # 11 have PWM capabilities. To output a PWM signal use the command: `analogWrite()`.



```
analogWrite(pin,value)
```

Parameters

pin: the number of the pin you want to write

value: the duty cycle between 0 (always off, 0%) and 255 (always on, 100%)

Returns

None

You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`.

4. Example

The following example reads an analog value from an analogy input pin, converts the value by dividing by 4, and outputs a PWM signal on a PWM pin (Fig. 2.2).

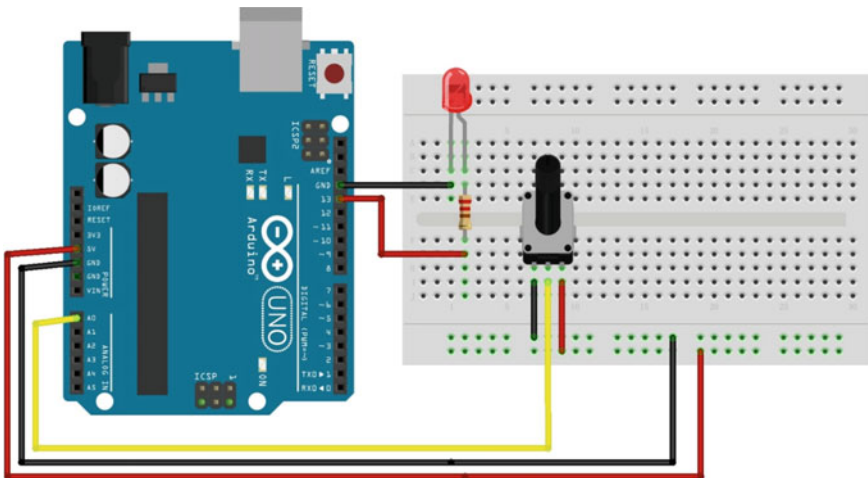


Fig. 2.2 Circuit layout for analogy signal read and write

```

1  int led = 13; // connect LED to pin 13
2  int pin = 0;  // potentiometer on analog pin 0
3  int value = 0; // variable to store the read value
4
5  void setup() {
6  }
7  void loop() {
8      value = analogRead(pin); // set value equal to the pin 0's input
9      value /= 4;               // converts 0-1023 to 0-255
10     analogWrite(led, value); // output PWM signal to LED
11 }
12

```

2.5 Advanced I/O Functions

1. `shiftOut(dataPin,clockPin,bitOrder,value)`

Shifts out a byte of data one bit at a time. Starts from either the most (i.e., the leftmost) or least (rightmost) significant bit. Each bit is written in turn to a data pin, after which a clock pin is pulsed (taken high, then low) to indicate that the bit is available. The syntax is as follows.



```
shiftOut(dataPin,clockPin,bitOrder,value)
```

Parameters

dataPin: the pin on which to output each bit (*int*)
clockPin: the pin to toggle once the dataPin has been set to the correct value (*int*)
bitOrder: which order to shift out the bits; either MSBFIRST or LSBFIRST. (Most Significant Bit First, or, Least Significant Bit First)
value: the data to shift out (*byte*)

Returns

None

This is known as synchronous serial protocol and is a common way that microcontrollers communicate with sensors, and with other microcontrollers. The two devices always stay synchronized, and communicate at close to maximum speeds, since they both share the same clock line. Often referred to as SPI (synchronous protocol interface) in hardware documentation.

2. `pulseIn(pin,value,timeout)`

Reads a pulse (either HIGH or LOW) on a pin. For example, if the value is HIGH, `pulseIn()` waits for the pin to go HIGH, starts timing, then waits for the pin to go LOW and stops timing. Returns the length of the pulse in microseconds. Gives up and returns 0 if no pulse starts within a specified time out. The timing of this function has been determined empirically and will probably show errors for longer pulses. Works on pulses from 10 μ s to 3 min in length. The syntax is as follows.



```
pulseIn(pin,value,timeout)
```

Parameters

`pin`: the number of the pin on which you want to read the pulse (*int*)

`value`: type type of pulse to read: either HIGH or LOW (*int*)

`timeout (optional)`: the number of microseconds to wait for the pulse to start; default is one second (*unsigned long*)

Returns

the length of the pulse (in microseconds) or 0 if no pulse started before the timeout

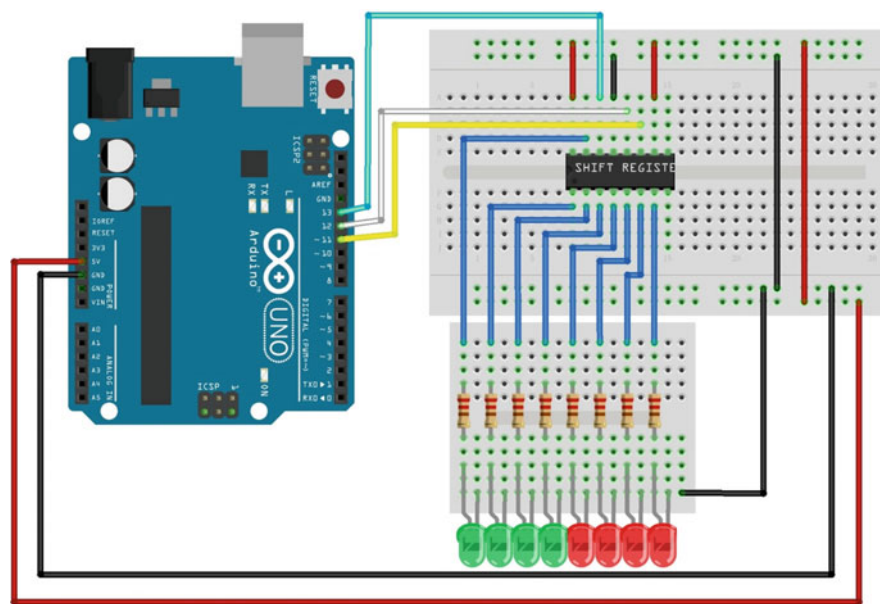


Fig. 2.3 Circuit layout for the shift register

Please go to Chap. 4, and refer the example for pulseIn().

3. Example

In this example, we will be using the 74HC595 8-bit shift register, which you can pick up from most places at a very reasonable price. This shift register will provide us with a total of eight extra pins to use. The layout is as follows (Fig. 2.3).

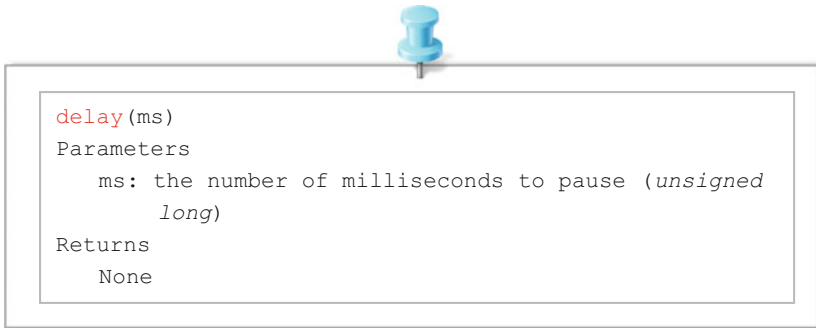
In this example, we increment the currentLED variable and pass it to the bitSet method. The bit is set to the left of the previous one to 1 every time, thereby informing the shift register to activate the output to the left of the previous one. As a result, the LEDs light up one by one.

```
1  int latchPin = 12;
2  int clockPin = 11;
3  int dataPin = 13;
4  byte leds = 0;
5  int currentLED = 0;
6  void setup() {
7      pinMode(latchPin, OUTPUT);
8      pinMode(dataPin, OUTPUT);
9      pinMode(clockPin, OUTPUT);
10     leds = 0;
11 }
12 void loop() {
13     leds = 0;
14     if (currentLED == 7) {
15         currentLED = 0;
16     }
17     else {
18         currentLED++;
19     }
20     bitSet(leds, currentLED);
21     digitalWrite(latchPin, LOW);
22     shiftOut(dataPin, clockPin, LSBFIRST, leds);
23     digitalWrite(latchPin, HIGH);
24     delay(250);
25 }
```

2.6 Timer Functions

1. **delay(ms)**

Pauses the program for the amount of time (in milliseconds) specified as the parameter. The syntax for the function is as follows.

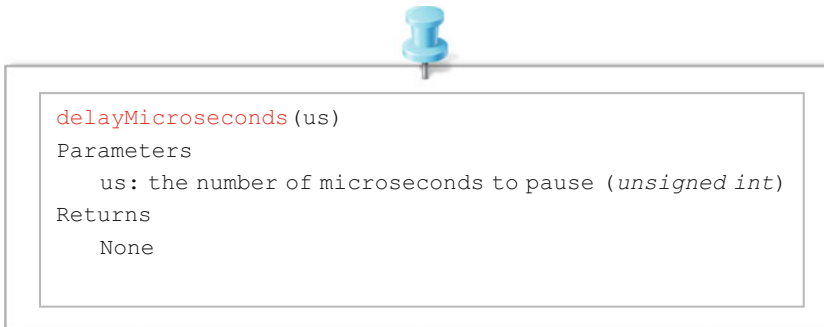


Time is specified in milliseconds, where a delay of 1000 ms equals 1 s, 2000 ms equals 2 s, and so on. This value can be expressed as a constant or variable in the unsigned long data type.

The use of `delay()` in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulations can function during the delay function, so in effect, it brings most other activities to a halt.

2. `delayMicroseconds(us)`

Rather than a long delay, the `delayMicroseconds()` function is used to delay for a much shorter time. The syntax is as follows:




Unlike `delay()`, time here is specified in microseconds, or millionths of a second, where a time period of 1000 μ s would equal 1 ms or 0.001 of a second, 10,000 would equal 10 ms or 0.01 of a second, and so on.

3. `millis()`

Inside the microcontroller on the Arduino board there are three onboard hardware timers that work in the background to handle repetitive tasks like incrementing counters or keeping track of program operations. Each of these timers is already being used in some capacity, usually for handling hardware PWM and system

timing. The `millis()` function makes use of one of these hardware timers to maintain a running counter of how many milliseconds the microcontroller has been running since the last time it was turned on or reset. Because this function uses a hardware timer, it performs its counting in the background with no impact on the flow or resources of our source code.




```
millis()  
Parameters  
    None  
Returns  
    Number of milliseconds since the program started  
    (unsigned long)
```

By calling the function, it returns a value in milliseconds that can be used like any other variable as part of a conditional test, to perform arithmetic operations, or to be assigned to other variables. Because this function returns a value in an unsigned long data type, it will overflow, or reset to 0, in about 50 days. It can also result in undesired problems if an expression is performed on it using other data types like integers.

4. `micros()`

Where the `millis()` function returns the current operating time in milliseconds, the `micros()` function does the same, but in microseconds. This could be used in exactly the same manner as `millis()`, just on a much smaller scale, effectively returning the value 1000 for every 1 that `millis()` would return.




```
micros()  
Parameters  
    None  
Returns  
    Number of microseconds since the program started  
    (unsigned long)
```

Unlike `millis()`, `micros()` will overflow, or reset back to 0, every 70 min.

2.7 Communication Functions

1. `Serial.begin(speed)`

Opens the serial port and sets the baud rate for serial data transmission. The typical baud rate for communicating with the computer is 9600, although other speeds are also supported, i.e., 300, 600, 1200, 2400, 4800, 9600, 14,400, 19,200, 28,800, 38,400, 57,600, or 115,200.




```
Serial.begin(speed)
Parameters
    speed: set the baud rate
Returns
    None
```

It should be noted that the digital pins 0 (RX) and 1 (TX) cannot be used at the same time when using serial communication.

2. `Serial.available()`

Receives the number of bytes (characters) available for reading from the serial port. This is data that has already arrived and been stored in the serial receive buffer.



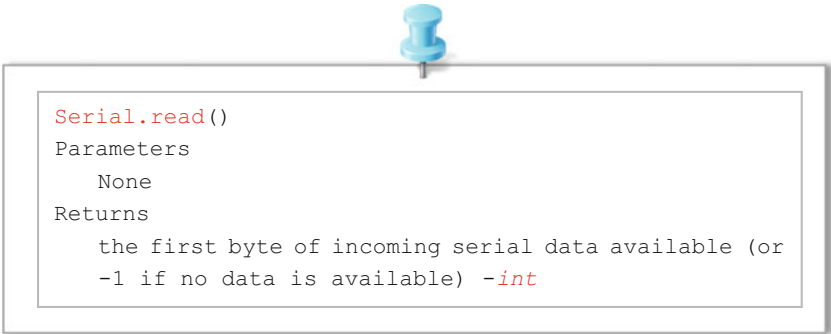
```
Serial.available()
Parameters
    None
Returns
    the number of bytes available to read
```

Remember, the hardware serial port on the Arduino microcontroller has a buffer that can store up to 128 bytes of information so that it is not lost. If no data is waiting for us, it will return 0. On the other hand, if any data is available, the function will return a value other than 0, which will signify true. We can proceed to read from the buffer.

`Serial.available()` inherits from the `Stream` utility class.

3. `Serial.read()`

Reads incoming serial data.



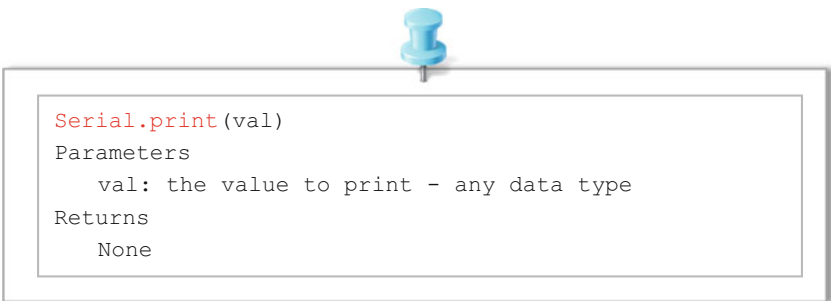
```
Serial.read()  
Parameters  
    None  
Returns  
    the first byte of incoming serial data available (or  
    -1 if no data is available) -int
```

This function simply returns the first byte of information available in the serial buffer. Because of the way our serial communications are structured, each character that we send to the Arduino through the Serial Monitor will be converted to that character's ASCII character value. For example, if we were to send the Arduino the number 1, instead of receiving the numerical integer 1, the Arduino will actually receive the numerical value 49 corresponding to that character's ASCII character code.

`Serial.read()` inherits from the `Stream` utility class

4. `Serial.print(val)`

Prints data to the serial port as human-readable ASCII text.

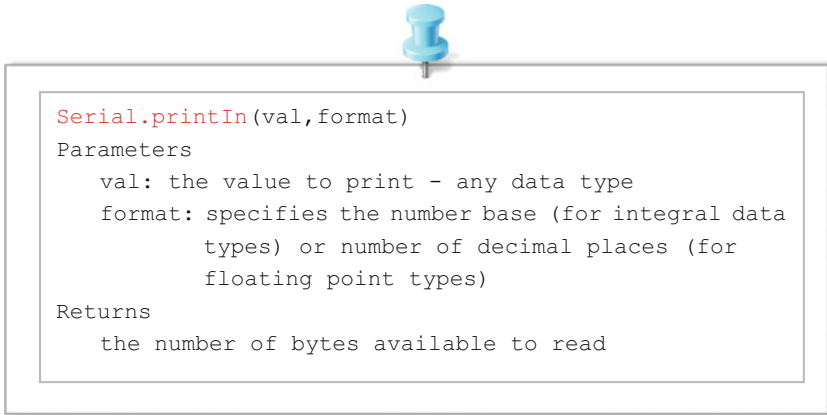


```
Serial.print(val)  
Parameters  
    val: the value to print - any data type  
Returns  
    None
```

This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character.

5. `Serial.println(val,format)`

Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13, or `\r`) and a newline character (ASCII 10, or `\n`).



The `println()` function is a little easier to use and helps to clean up the output that we receive from the Serial Monitor. You will often see both the `print()` and `println()` functions used in conjunction to format the output, making the text easier to read.

6. Example

In this example, two Arduinos are used. The Arduino Uno on the left is our sender and the Arduino Mega on the right is our receiver. We use the Mega to make it easier to display debug information on the computer. The Arduinos are connected together using digital 0 and 1 (RX and TX) on the Uno and digital 16 and 17 (RX2 and TX2) on the Mega. The receiver on one needs to be connected to the transmit of the other, and vice versa. The Arduinos also need to have a common reference between the two. This is ensured by running a ground wire (Fig. 2.4).

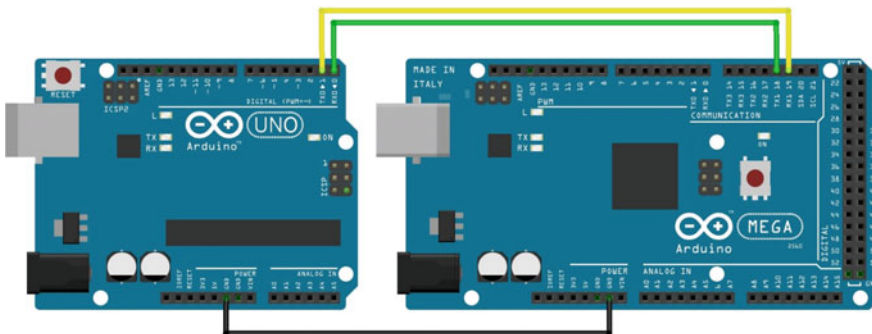


Fig. 2.4 The schematic of serial communication

The first step is to package the string to be communicated. In general, a packet is comprised of some start byte, a payload (the data you wish to send), and a checksum to validate your data. Here, the packet is: $[0 \times 53] + [\text{counter value}] + [\text{static value}] + [\text{checksum}]$.

Sender Code

The simple sender coder below increments our counter and sends our packet.

```
1  // Sender Information
2  unsigned char START_BYTE = 0x53; // ASCII "S"
3  unsigned char counterValue = 0;
4  unsigned char staticValue = 5;
5  unsigned char checksum = 0;
6
7  void setup() {
8      Serial.begin(9600);
9  }
10 void loop() {
11     // Increment our counter
12     counterValue = counterValue + 1;
13     // Check for overflow, and loop
14     if (counterValue > 250)
15         counterValue = 0;
16     // Calculate our checksum
17     checksum = counterValue + staticValue;
18     // Important: Serial.write must be used, not print
19     Serial.write(START_BYTE);
20     Serial.write(counterValue);
21     Serial.write(staticValue);
22     Serial.write(checksum);
23     // We only need to send a packet every 250ms.
24     // If your code starts to get complicated,
25     // consider using a timer instead of a delay
26     delay(250);
27 }
```

Receiver Code

For the receiver code, we constantly go through the main loop and check whether we have information ready to be read. Once, we receive our first byte we compare it to our expected start byte. If this passes, then we set a flag and wait for the rest of the packet to roll in. Once, we have the expected packet then we read the values in it, calculate our checksum, and then print out the result on our terminal.

```
1 // Receiver Information
2 unsigned char START_BYTE = 0x53; // ASCII "S"
3 unsigned char counterValue = 0;
4 unsigned char staticValue = 0;
5 unsigned char checksum = 0;
6 boolean syncByteFound = 0; // Sync Byte flag
7
8 void setup() {
9     Serial.begin(9600);
10    Serial2.begin(9600);
11 }
12
13 void loop() {
14     unsigned char rxByte = 0;
15     unsigned char calculatedChecksum = 0;
16     // Check to see if there's something to read
17     if (Serial2.available() > 0) {
18         // If we're waiting for a new packet, check for the sync byte
19         if (syncByteFound == 0) {
20             rxByte = Serial2.read();
21             if (rxByte == 0x53)
22                 syncByteFound = 1;
23         }
24         // If we've found our sync byte, check for expected number of
25 bytes
26         if (Serial2.available() > 2) {
27             counterValue = Serial2.read();
28             staticValue = Serial2.read();
29             checksum = Serial2.read();
30             calculatedChecksum = counterValue + staticValue;
31             // Print out our serial information to debug
32             Serial.print("[");
33             Serial.print("S");
34             Serial.print("]");
35             Serial.print("[");
36             Serial.print(counterValue);
37             Serial.print("]");
38             Serial.print("[");
39             Serial.print(staticValue);
40             Serial.print("]");
41             Serial.print("[");
```

```
42     Serial.print(checksum);
43     Serial.print(" ");
44     if (calculatedChecksum == checksum)
45         Serial.println("[Checksum Passed]");
46     else
47         Serial.println("[Checksum FAILED]");
48     syncByteFound = 0;
49 }
50 }
51 }
52
```

2.8 Interrupt Functions

1. `attachInterrupt(digitalPinToInterrupt(pin),ISR,mode)`

The `attachInterrupt()` function enables hardware interrupts and links a hardware pin to an ISR to be called when the interrupt is triggered. This function also specifies the type of state change that will trigger the interrupt. Its syntax is as follows:



```
attachInterrupt(digitalPinToInterrupt(pin),ISR,mode)
```

Parameters

`interrupt`: the number of the interrupt (int)

`pin`: the pin number

`ISR`: the interrupt service routine (ISR) to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.

`mode`: defines when the interrupt should be triggered. Four constants are predefined as valid values:

`LOW` to trigger the interrupt whenever the pin is low,

`CHANGE` to trigger the interrupt whenever the pin changes value

`RISING` to trigger when the pin goes from low to high,

`FALLING` for when the pin goes from high to low.

Returns

None

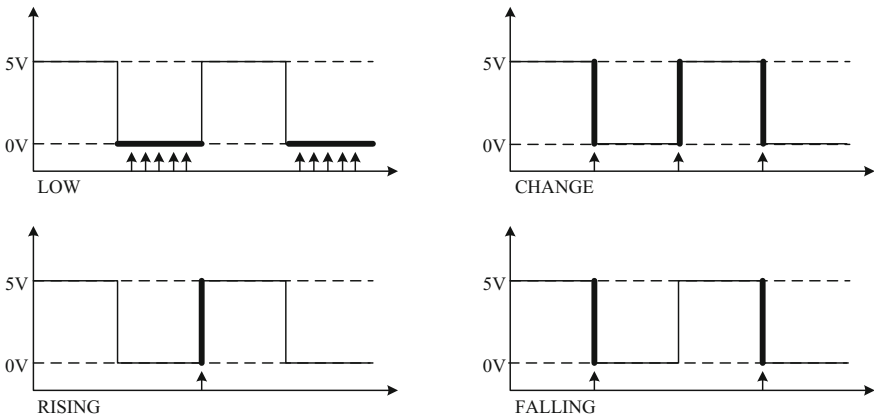


Fig. 2.5 State changes

Normally, you should use `digitalPinToInterrupt(pin)`, rather than place an interrupt number directly into your sketch. The specific pins with interrupts, and their mapping to interrupt numbers vary for each type of board. On the DFrobot Romeo board, there are two possible hardware interrupts, 0 (for digital pin 2) and 1 (for digital pin 3).

Four possible modes are shown in Fig. 2.3, which include LOW, CHANGE, RISING, and FALLING (Fig. 2.5).

2. `detachInterrupt(interrupt)`

In some cases, we might need to change the mode of an enabled interrupt. For example, we may change the mode from RISING to FALLING. For this, we need to first stop the interrupt by using the `detachInterrupt()` function. Its syntax is as follows:

`detachInterrupt(interrupt)`

Parameters

interrupt: the number of the interrupt to disable


Returns

None

With only one parameter to determine which interrupt we are disabling, this parameter is specified as either 0 or 1. Once the interrupt has been disabled, we can then reconfigure it using a different mode in the `attachInterrupt()` function.

3. `interrupts()`

All interrupts in Arduino can be enabled by the function `interrupts()`. The syntax is as follows:




```
interrupts()  
Parameters  
    None  
Returns  
    None
```

Interrupts allow certain important tasks to run in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of a code, however, and may be disabled for particularly critical sections of code.

4. `noInterrupts()`

To deactivate all interrupts, we can use the function `noInterrupts()`. The syntax is as follows.



```
noInterrupts()  
Parameters  
    None  
Returns  
    None
```

5. Example

In this example, we blink the built-in LED every 500 ms, during which time both interrupt pins are monitored. When the button on the interrupt 0 is pressed, the value for `micros()` is displayed on the Serial Monitor, and when the button on the interrupt 1 is pressed, the value for `millis()` is displayed (Fig. 2.6).

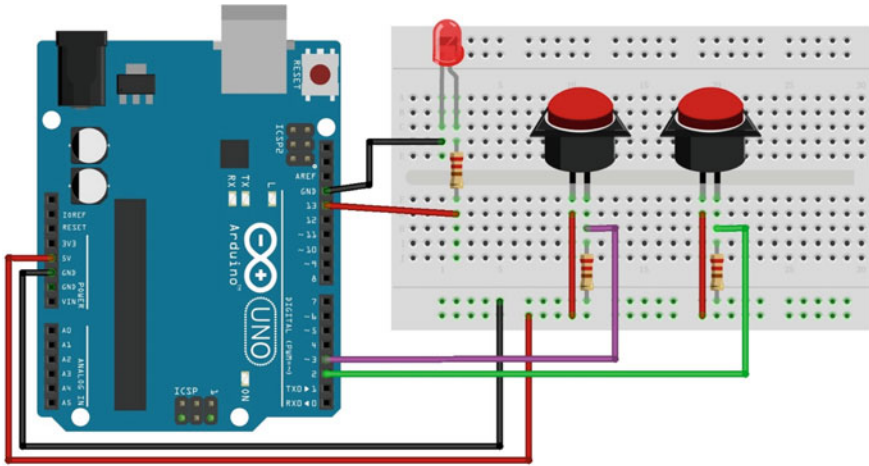


Fig. 2.6 The circuit for interrupts and time functions

```

1  #define LED 13
2  void setup() {
3      Serial.begin(9600);
4      pinMode(LED, OUTPUT);
5      attachInterrupt(0, displayMicros, RISING);
6      attachInterrupt(1, displayMillis, RISING);
7  }
8  void loop() {
9      digitalWrite(LED, HIGH);
10     delay(500);
11     digitalWrite(LED, LOW);
12     delay(500);
13 }
14 void displayMicros() {
15     Serial.write("micros()=");
16     Serial.println(micros());
17 }
18 void displayMillis() {
19     Serial.write("millis()=");
20     Serial.println(millis());
21 }

```

2.9 Math Functions

The Arduino language supports many math functions which make it possible to perform sophisticated data processing in a data logger sketch that would otherwise have to be performed offline in some other software application.

1. **min(x,y)**

Calculates the minimum of two numbers



```
min(x,y)
```

Parameters

x: the first number, any data type

y: the second number, any data type

Returns

The smaller of the two numbers

2. **max(x,y)**

Calculates the maximum of two numbers.



```
max(x,y)
```

Parameters

x: the first number, any data type

y: the second number, any data type

Returns

The larger of the two numbers

3. **abs(x)**

Computes the absolute value of a number



```
abs(x)
Parameters
  x: the number
Returns
  x: if x is greater than or equal to 0.
  -x: if x is less than 0.
```

Because of the way the `abs()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results.

4. **Trigonometric functions**

The trigonometric functions include `sin(rad)`, `cos(rad)`, and `tan(rad)`. All trigonometric functions accept as input and return as output angles in radians, not degrees: $\text{radians} = \text{degrees} \times \pi/180$ and vice versa to convert radians back to degrees.

5. **pow(base,exponent)**

Calculates the value of a number raised to a power. `pow()` can be used to raise a number to a fractional power. This is useful for generating exponential mapping of values or curves.



```
pow(base,exponent)
Parameters
  base: the number (float)
  exponent: the power to which the base is raised (float)
Returns
  The result of the exponentiation (double)
```


6. **sqrt(x)**

Calculates the square root of a number.



```
sqrt(x)
Parameters
    x: the number, any data type
Returns
    double, the number's square root
```

7. **constrain(x,a,b)**

Constrains a number to be within a range.



```
constrain(x,a,b)
Parameters
    x: the number to constrain, all data types
    a: the lower end of the range, all data types
    b: the upper end of the range, all data types
Returns
    x: if x is between a and b
    a: if x is less than a
    b: if x is greater than b
```

8. **map(value,fromLow,fromHigh,toLow,toHigh)**

Remaps a number from one range to another. That is, a value of fromLow would be mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.



```
map(value,fromLow,fromHigh,toLow,toHigh)
```

Parameters

value: the number to map

fromLow: the lower bound of the value's current range

fromHigh: the upper bound of the value's current range

toLow: the lower bound of the value's target range

toHigh: the upper bound of the value's target range

Returns

The mapped value

9. **random(min,max)**

The random() function returns a semi-random number up to the parameters specified. If no parameters are specified, it will return a value in the signed long data type, with a range of -2,147,483,648-2,147,483,647. Its syntax is as follows:



```
random(min,max)
```

Parameters

min: lower bound of the random value, inclusive (optional)

max: upper bound of the random value, exclusive

Returns

a random number between min and max

The random() function will assume a value of 0 as its minimum value.

10. Example

In this example, we create a sine wave and configure the brightness of the LED to follow the path of the wave. This is what makes the light pulsate in the form of a sine wave instead of just illuminating up to full brightness and back down again (Fig. 2.7).

The codes are as follows (Fig. 2.7)

```

1  int ledPin = 11;
2  float sinVal;
3  int ledVal;
4  void setup() {
5      pinMode(ledPin, OUTPUT);
6  }
7  void loop() {
8      for (int x = 0; x < 180; x++) {
9          // convert degrees to radians
10         // then obtain sin value
11         sinVal = (sin(x * (3.1412 / 180)));
12         ledVal = int(sinVal * 255);
13         analogWrite(ledPin, ledVal);
14         delay(25);
15     }
16 }

```

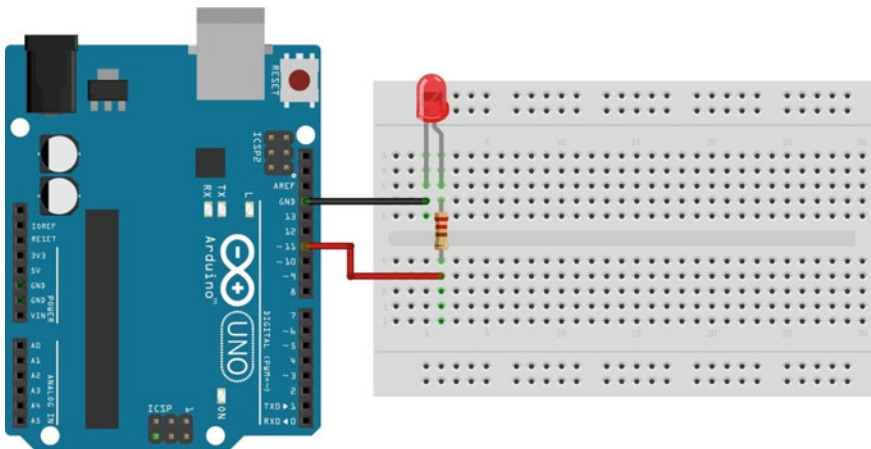


Fig. 2.7 The circuit for mathematical example

Table 2.1 Programming language reference

Construct	Description
int	Integer values, such as 123
float	Decimal values, such as 1.15
char[]	String values, such as “Arduino”
HIGH	Digital pin with current
LOW	Digital pin with no current
INPUT	Pin can only be read
OUTPUT	Pin can only be set
A0–A7	Constants for analog pins; varies by board
0–13	Value for digital pins; varies by board
analogRead()	Returns analog pin value (0–1023)
analogWrite()	Sets analog pin value
digitalRead()	Returns digital pin value (HIGH or LOW)
digitalWrite()	Sets digital pin value (HIGH or LOW)
Serial.begin()	Initializes serial monitor
Serial.print()	Logs message on serial monitor
Serial.println()	Logs message on serial monitor with new line
delay(ms)	Adds a wait in processing
setup()	Standard Arduino function called once
loop()	Standard Arduino function called repeatedly
if	Checks for a true/false condition
if ... else	Checks for a true/false condition; if false goes to else
//	Single-line comment
/* */	Multiline comment
#define	Defines a constant
#include	Includes an external library

2.10 Programming Language Reference

The Arduino programming language has a number of constructs. Here, we just provide the basics that have been used in this book (see Table 2.1). You can explore the complete language at <https://www.arduino.cc/en/Reference>.

Designing Embedded Systems with Arduino

A Fundamental Technology for Makers

Pan, T.; Zhu, Y.

2018, XVI, 228 p. 139 illus., 103 illus. in color.,

Hardcover

ISBN: 978-981-10-4417-5