Welcome!

Thank you for purchasing our *AZ-Delivery 1.3 inch OLED I2C Screen*. On the following pages, you will be introduced to how to use and set up this handy device.

Have fun!

Table of Contents

Introduction

OLED stands for Organic Light Emitting Diodes. OLED screens are arrays of LEDs stacked together in a matrix. The 1.3 OLED screen has 128x64 pixels (LEDs). To control these LEDs we need a driver circuit or a chip. The screen has a driver chip called *SH1106*. The driver chip has an I2C interface for communication with the main microcontroller.

The OLED screen and SH1106 driver chip operate in the *3.3V* range. But there is an on-board *3.3V* voltage regulator, therefore, these screens can be operated in the *5V* range.

The performance of these screens is much better than traditional LCDs. Simple I2C communication and low power consumption make them more suited for a variety of applications.

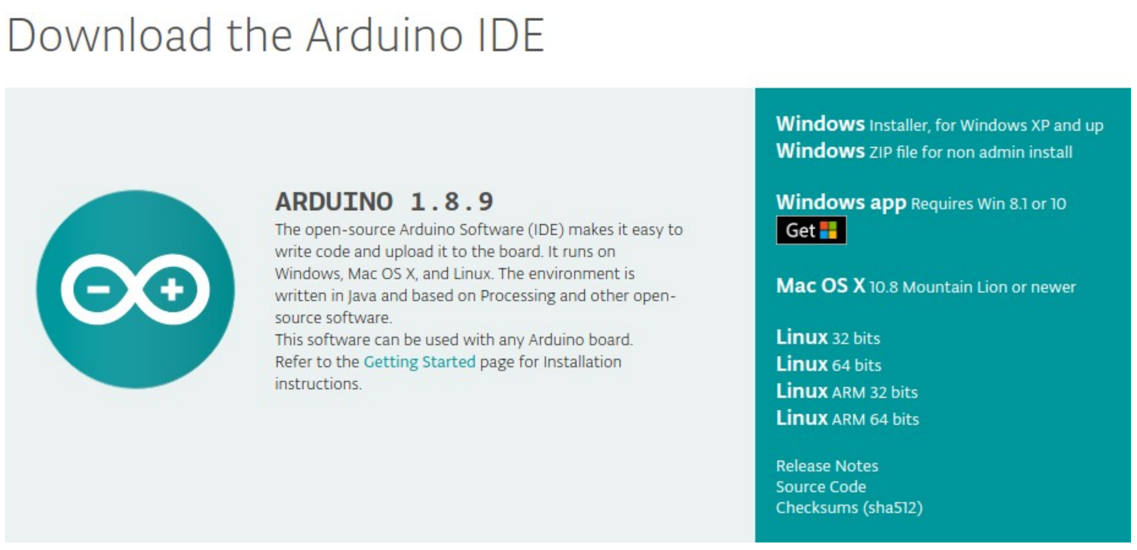
Specifications

Power supply voltage	from 3.3V to 5V
Communication interface	I2C
Pixel Color	White
Operating temperature	from -20 to 70 °C
Low power consumption	< 11mA
Dimensions	36 x 34 x 3mm [1.4 x1.3 x 0.1inch]

To extend the lifetime of the screen, it is common to use a “Screen saver”*.* It is recommended not to use constant information over a long period of time, because that will shorten the lifespan of the screen and increase the, so called, “Screen burn” effect.

How to set-up Arduino IDE

If the Arduino IDE is not installed, follow the *link* and download the installation file for the operating system of choice.


Download the Arduino IDE

ARDUINO 1.8.9

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software.
This software can be used with any Arduino board.
Refer to the Getting Started page for Installation instructions.

Windows Installer, for Windows XP and up
Windows ZIP file for non admin install

Windows app Requires Win 8.1 or 10
Get

Mac OS X 10.8 Mountain Lion or newer

Linux 32 bits
Linux 64 bits
Linux ARM 32 bits
Linux ARM 64 bits

Release Notes
Source Code
Checksums (sha512)

For `Windows` users, double click on the downloaded *.exe* file and follow the instructions in the installation window.

For *Linux* users, download a file with the extension *.tar.xz*, which has to be extracted. When it is extracted, go to the extracted directory and open the terminal in that directory. Two *.sh* scripts have to be executed, the first called *arduino-linux-setup.sh* and the second called *install.sh*.

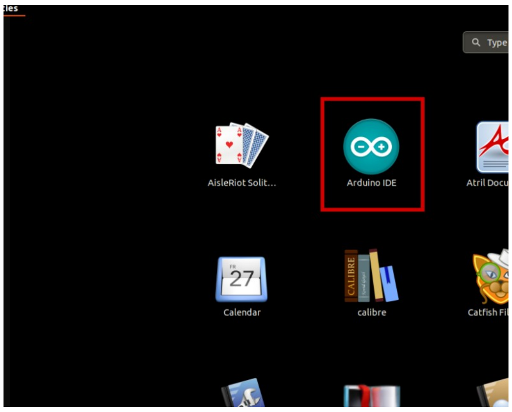
To run the first script in the terminal, open the terminal in the extracted directory and run the following command:

```
sh arduino-linux-setup.sh user_name
```

user_name - is the name of a superuser in the Linux operating system. A password for the superuser has to be entered when the command is started. Wait for a few minutes for the script to complete everything.

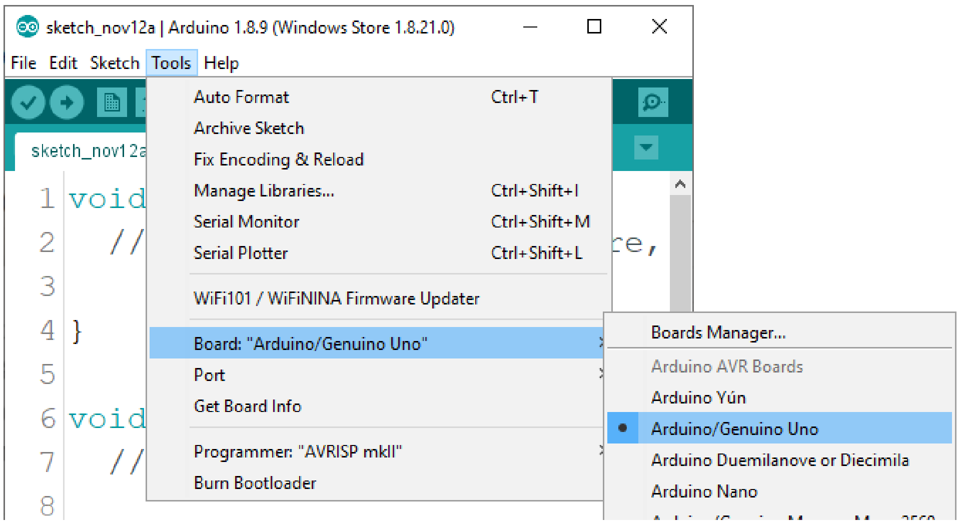
The second script called *install.sh* script has to be used after installation of the first script. Run the following command in the terminal (extracted directory): **sh install.sh**

After the installation of these scripts, go to the *All Apps*, where the *Arduino IDE* is installed.



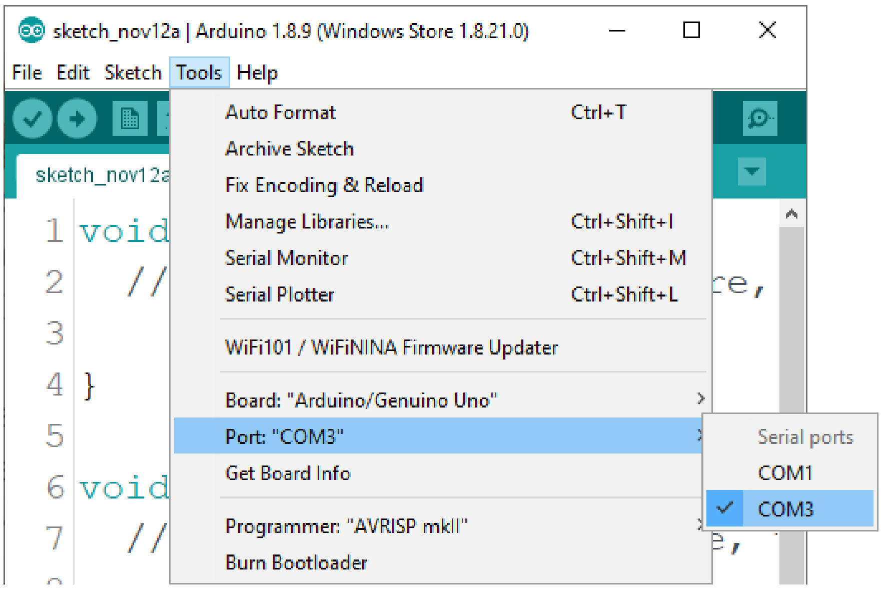
Almost all operating systems come with a text editor preinstalled (for example, *`Windows`* comes with *`Notepad`*, *`Linux Ubuntu`* comes with *`Gedit`*, *`Linux Raspbian`* comes with *`Leafpad`*, etc.). All of these text editors are perfectly fine for the purpose of the eBook.

Next thing is to check if your PC can detect an Arduino board. Open freshly installed Arduino IDE, and go to:
Tools > Board > {your board name here}
{your board name here} should be the *`Arduino/Genuino Uno`*, as it can be seen on the following image:


sketch_nov12a | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino/Genuino Uno"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Boards Manager...
Arduino AVR Boards
Arduino Yún
Arduino/Genuino Uno
Arduino Duemilanove or Diecimila
Arduino Nano

The port to which the Arduino board is connected has to be selected. Go to:
Tools > Port > {port name goes here}
and when the Arduino board is connected to the USB port, the port name can be seen in the drop-down menu on the previous image.

If the Arduino IDE is used on Windows, port names are as follows:


sketch_nov12a | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino/Genuino Uno"
Port: "COM3"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM1
COM3

For `Linux` users, for example port name is */dev/ttyUSBx*, where *x* represents integer number between *0* and *9*.

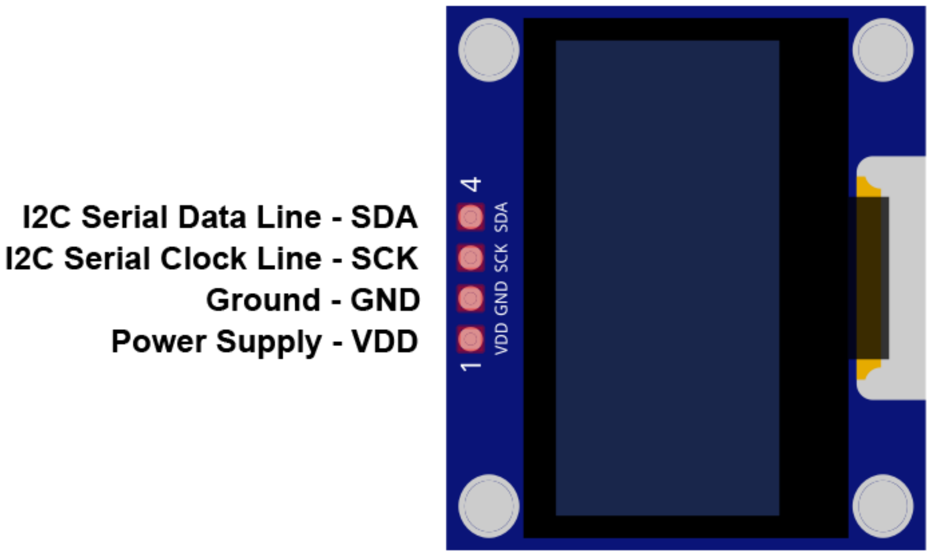
How to set-up the Raspberry Pi and Python

For the Raspberry Pi, first the operating system has to be installed, then everything has to be set-up so that it can be used in the *Headless* mode. The *Headless* mode enables remote connection to the Raspberry Pi, without the need for a *PC* screen Monitor, mouse or keyboard. The only things that are used in this mode are the Raspberry Pi itself, power supply and internet connection. All of this is explained minutely in the free eBook:
Raspberry Pi Quick Startup Guide

The *Raspbian* operating system comes with *Python* preinstalled.

The pinout

The 1.3 inch OLED screen has four pins. The pinout is shown on the following image:

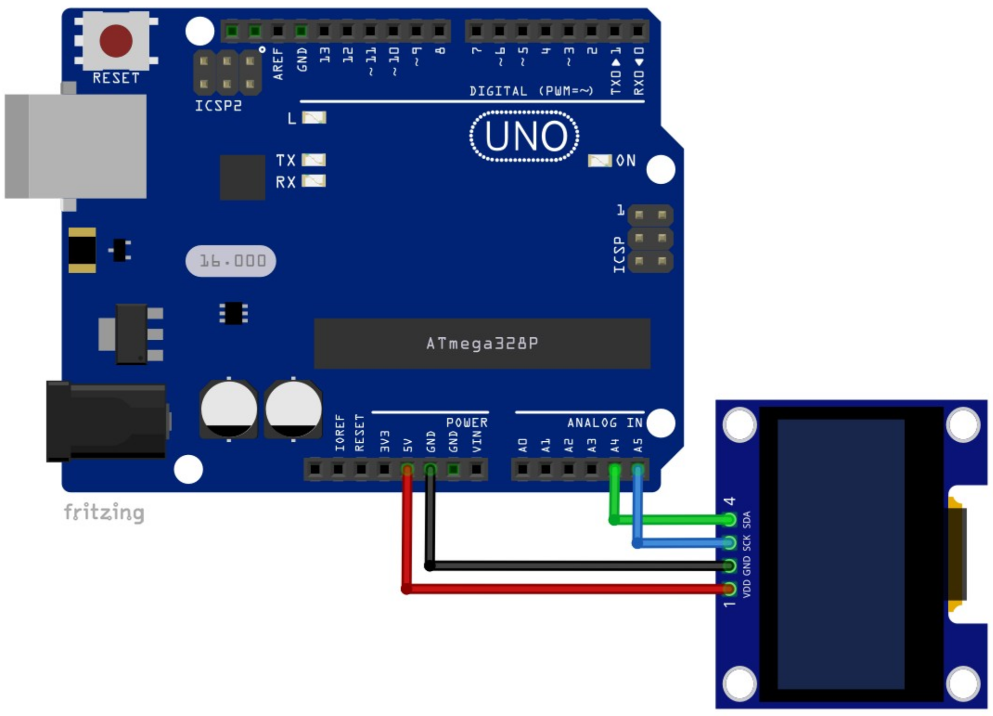


The screen has an on-board voltage regulator. The pins of the *1.3* inch OLED screen can be connected to *3.3V* or to *5V* power supply without danger to the sensor itself.

NOTE: When using Raspberry Pi, the power supply should be drawn from a 3.3V pin only.

Connecting the screen with Uno

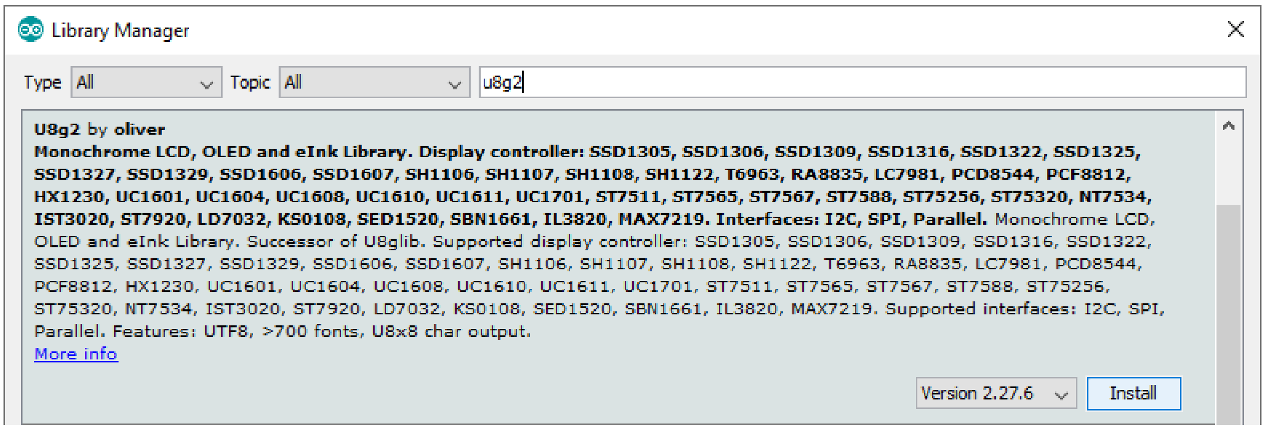
Connect the 1.3 inch OLED screen with the Uno as shown on the following connection diagram:



Screen pin	Uno pin	Wire color
SDA	A4	**Green wire**
SCK	A5	**Blue wire**
GND	GND	**Black wire**
VCC	5V	**Red wire**

Library for Arduino IDE

To use the screen with an Uno, it is recommended to download an external library for it. The library that is going to be used is called the "*U8g2*". To download and install it, open Arduino IDE and go to: *Tools > Manage Libraries*. When a new window opens, type "*u8g2*" in the search box and install the library "*U8g2*" made by "*oliver*", as shown in the following image:



U8g2 by **oliver**
Monochrome LCD, OLED and eInk Library. Display controller: SSD1305, SSD1306, SSD1309, SSD1316, SSD1322, SSD1325, SSD1327, SSD1329, SSD1606, SSD1607, SH1106, SH1107, SH1108, SH1122, T6963, RA8835, LC7981, PCD8544, PCF8812, HX1230, UC1601, UC1604, UC1608, UC1610, UC1611, UC1701, ST7511, ST7565, ST7567, ST7588, ST75256, ST75320, NT7534, IST3020, ST7920, LD7032, KS0108, SED1520, SBN1661, IL3820, MAX7219. Interfaces: I2C, SPI, Parallel. Monochrome LCD, OLED and eInk Library. Successor of U8glib. Supported display controller: SSD1305, SSD1306, SSD1309, SSD1316, SSD1322, SSD1325, SSD1327, SSD1329, SSD1606, SSD1607, SH1106, SH1107, SH1108, SH1122, T6963, RA8835, LC7981, PCD8544, PCF8812, HX1230, UC1601, UC1604, UC1608, UC1610, UC1611, UC1701, ST7511, ST7565, ST7567, ST7588, ST75256, ST75320, NT7534, IST3020, ST7920, LD7032, KS0108, SED1520, SBN1661, IL3820, MAX7219. Supported interfaces: I2C, SPI, Parallel. Features: UTF8, >700 fonts, U8x8 char output.
More info

Several sketch examples come with the library, to open one, go to:
File > Examples > U8g2 > full_buffer > GraphicsTest
With this sketch example, you can test your screen. However, the code used in the example is fairly complex. The sketch is modified to make a more beginner-friendly version of the code.

Sketch example

```
#include <U8g2lib.h>
#include <Wire.h>
#define time_delay 2000
U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(U8G2_R0, U8X8_PIN_NONE);

const char COPYRIGHT_SYMBOL[] = {0xa9, '\0'};
void u8g2_prepare() {
  u8g2.setFont(u8g2_font_6x10_tf);
  u8g2.setFontRefHeightExtendedText();
  u8g2.setDrawColor(1);
  u8g2.setFontPosTop();
  u8g2.setFontDirection(0);
}
void u8g2_box_frame() {
  u8g2.drawStr(0, 0, "drawBox");
  u8g2.drawBox(5, 10, 20, 10);
  u8g2.drawStr(60, 0, "drawFrame");
  u8g2.drawFrame(65, 10, 20, 10);
}
void u8g2_r_frame_box() {
  u8g2.drawStr(0, 0, "drawRFrame");
  u8g2.drawRFrame(5, 10, 40, 15, 3);
  u8g2.drawStr(70, 0, "drawRBox");
  u8g2.drawRBox(70, 10, 25, 15, 3);
}
void u8g2_disc_circle() {
  u8g2.drawStr(0, 0, "drawDisc");
  u8g2.drawDisc(10, 18, 9);
  u8g2.drawStr(60, 0, "drawCircle");
  u8g2.drawCircle(70, 18, 9);
}
```

```
void u8g2_string_orientation() {
  u8g2.setFontDirection(0);
  u8g2.drawStr(5, 15, "0");
  u8g2.setFontDirection(3);
  u8g2.drawStr(40, 25, "90");
  u8g2.setFontDirection(2);
  u8g2.drawStr(75, 15, "180");
  u8g2.setFontDirection(1);
  u8g2.drawStr(100, 10, "270");
}
void u8g2_line() {
  u8g2.drawStr(0, 0, "drawLine");
  u8g2.drawLine(7, 20, 77, 32);
}
void u8g2_triangle() {
  u8g2.drawStr(0, 0, "drawTriangle");
  u8g2.drawTriangle(14, 20, 45, 30, 10, 32);
}
void u8g2_unicode() {
  u8g2.drawStr(0, 0, "Unicode");
  u8g2.setFont(u8g2_font_unifont_t_symbols);
  u8g2.setFontPosTop();
  u8g2.setFontDirection(0);
  u8g2.drawUTF8(10, 20, "☀");
  u8g2.drawUTF8(30, 20, "☁");
  u8g2.drawUTF8(50, 20, "☂");
  u8g2.drawUTF8(70, 20, "☔");
  u8g2.drawUTF8(95, 20, COPYRIGHT_SYMBOL); //COPYRIGHT SIMBOL
  u8g2.drawUTF8(115, 15, "\xb0"); // DEGREE SYMBOL
}
```

```
#define image_width 128
#define image_height 21
static const unsigned char image_bits[] U8X8_PROGMEM = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x06, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xfc, 0x1f, 0x00, 0x00,
  0xfc, 0x1f, 0x00, 0x00, 0x06, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xfe, 0x1f, 0x00, 0x00, 0xfc, 0x7f, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x07, 0x18, 0x00, 0x00, 0x0c, 0x60, 0x00, 0x00,
  0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x18, 0x00, 0x00,
  0x0c, 0xc0, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x03, 0x18, 0x00, 0x00, 0x0c, 0xc0, 0xf0, 0x1f, 0x06, 0x63, 0x80, 0xf1,
  0x1f, 0xfc, 0x33, 0xc0, 0x03, 0x18, 0x00, 0x00, 0x0c, 0xc0, 0xf8, 0x3f,
  0x06, 0x63, 0xc0, 0xf9, 0x3f, 0xfe, 0x33, 0xc0, 0x03, 0x18, 0x00, 0x00,
  0x0c, 0xc0, 0x18, 0x30, 0x06, 0x63, 0xc0, 0x18, 0x30, 0x06, 0x30, 0xc0,
  0xff, 0xff, 0xdf, 0xff, 0x0c, 0xc0, 0x18, 0x30, 0x06, 0x63, 0xe0, 0x18,
  0x30, 0x06, 0x30, 0xc0, 0xff, 0xff, 0xdf, 0xff, 0x0c, 0xc0, 0x98, 0x3f,
  0x06, 0x63, 0x60, 0x98, 0x3f, 0x06, 0x30, 0xc0, 0x03, 0x18, 0x0c, 0x00,
  0x0c, 0xc0, 0x98, 0x1f, 0x06, 0x63, 0x70, 0x98, 0x1f, 0x06, 0x30, 0xc0,
  0x03, 0x18, 0x06, 0x00, 0x0c, 0xc0, 0x18, 0x00, 0x06, 0x63, 0x38, 0x18,
  0x00, 0x06, 0x30, 0xc0, 0x03, 0x18, 0x03, 0x00, 0x0c, 0xe0, 0x18, 0x00,
  0x06, 0x63, 0x1c, 0x18, 0x00, 0x06, 0x30, 0xc0, 0x00, 0x80, 0x01, 0x00,
  0xfc, 0x7f, 0xf8, 0x07, 0x1e, 0xe3, 0x0f, 0xf8, 0x07, 0x06, 0xf0, 0xcf,
  0x00, 0xc0, 0x00, 0x00, 0xfc, 0x3f, 0xf0, 0x07, 0x1c, 0xe3, 0x07, 0xf0,
  0x07, 0x06, 0xe0, 0xcf, 0x00, 0x60, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x30, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0,
  0x00, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0xe0, 0x00, 0xfc, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7f, 0x00, 0xfc, 0x1f, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3f };
```

```
void u8g2_bitmap() {
  u8g2.drawXBMP(0, 5, image_width, image_height, image_bits);
}
void setup(void) {
  u8g2.begin();
  u8g2_prepare();
}
float i = 0.0;
void loop(void) {
  u8g2.clearBuffer();
  u8g2_prepare();
  u8g2_box_frame();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_disc_circle();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_r_frame_box();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_prepare();
  u8g2_string_orientation();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_line();
  u8g2.sendBuffer();
  delay(time_delay);
  // one tab
  u8g2.clearBuffer();
  u8g2_triangle();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_prepare();
  u8g2_unicode();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2_bitmap();
  u8g2.sendBuffer();
  delay(time_delay);

  u8g2.clearBuffer();
  u8g2.setCursor(0, 0);
  u8g2.print(i);
  i = i + 1.5;
  u8g2.sendBuffer();
  delay(time_delay);
}
```

First, two libraries, the *U8g2lib* and *Wire* are imported. Next, an object called *u8g2* is created with the following line of code:

```
U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_R0, U8X8_PIN_NONE);
```

The created object represents the screen itself and it is used to control the screen. The *U8g2* library can be used for many other OLED screens, thus there are many constructors in the sketch examples from the library.

Next, a function called *u8g2_prepare()* is created, which has no arguments and returns no value. The five *u8g2* library functions are used.

The first function is called *setFont()*, which has one argument and returns no value. The argument represents the *u8g2* font. Follow the link to see the list of availale fonts:
https://github.com/olikraus/u8g2/wiki/fntlist8x8

The second function is called *setFontRefHeightExtendedText()*, which has no arguments and returns no value. It is used for drawing characters on the screen. For more detailed explanation, follow the link:
https://github.com/olikraus/u8g2/wiki/u8g2reference#setfontrefheightextendedtext

The third function is called *`setDrawColor()`*, which has one argument and returns no value. The argument value is an integer number which represents a color index for all drawing functions. Font drawing procedures use this argument to set the foreground color. The default value is *1.* If it is set to *0* then the space around the character is lit up, and the character is not. Argument value *2* can be used also, but there's no difference from *0*.

The fourth function is called *`setFontPosTop()`*, which has no argument and returns no value. This function controls the character position in one line of text. The function has a couple of versions. The first is *`setFontPosBaseLine()`* second is *`setFontPosCenter()`*, and third is *`setFontPosBottom()`*.Their purpose is to change the position of the characters in the one line.

The fifth function is called *`setFontDirection()`*, which has one argument and returns no value. The argument is an integer number which represents direction of the text. The value is an integer number in a range from *0* to *3* (*0* = *0*°, *1* = *90*°, *2* = *180*° and *3* = *270*°).

The function called *`drawStr()`* has three arguments and returns no value. It is used to display a constant string on the screen. The first two arguments represent the *X* and *Y* positions of the cursor, where the text is displayed. The third argument represents the text itself, a string value. Functions that set-up text layout should be used before using *`drawStr()`* function, or else the *`drawStr()`* function will use the default settings for the font, size and overall layout of the text.

To display shapes, specific library functions for each shape has to be used:

The function called *`drawFrame()`* has four arguments and returns no value. It is used to display frame, an empty rectangle. The first two arguments represent the *X* and *Y* positions of the top left corner of the frame. The third argument represents the width of the frame and the fourth argument represents the height of the frame.

The function called *`drawRFrame()`* has five arguments and returns no value. It is used to display a frame with rounded corners. The first two arguments represent the *X* and *Y* positions of the top left corner of the frame. The second two arguments represent the width and height of the frame and the fifth argument represents the corner radius.

The function called *drawBox()* has four arguments and returns no value. It is used to display a filled rectangle. The first two arguments represent the *X* and *Y* positions of the top left corner of the rectangle. The second two arguments represent the width and height of the rectangle.

The function called *drawRBox()* has five arguments and returns no value. It is used to display a filled rectangle with rounded edges. The first two arguments represent the *X* and *Y* positions of the top left corner of the rectangle. The second two arguments represent the width and height of the rectangle and the fifth argument represents the corner radius.

The function called *drawCircle()* has three arguments and returns no value. It is used to display a circle. The first two arguments represent the *X* and *Y* positions of the circle center point. The third argument represents the circle radius.

The function called *drawDisc()* has three arguments and returns no value. It is used to display a disc. The first two arguments represent *X* and *Y* positions of the disc center point. The third argument represents the disc radius.

The function called *drawTriangle()* has six arguments and returns no value. It is used to display a filled triangle. The first two arguments represent the *X* and *Y* positions of the first corner point of the triangle. The second two arguments represent the *X* and *Y* positions of the second corner point of the triangle. The last two arguments represent the *X* and *Y* positions of the last corner point of the triangle.

The function called *drawLine()* has four arguments and returns no value. It is used to display a line. The first two arguments represent the *X* and *Y* positions of the starting point of the line. The second two arguments represent *X* and *Y* positions of the end point of the line.

The function called *drawUTF8()* has three arguments and returns a value. It is used to display a text, the string value that can contain a character encoded as a *Unicode* character. The first two arguments represent the *X* and *Y* positions of the cursor and the third represents the text itself. The *Unicode* characters can be displayed in a couple of ways. The first is to copy and paste the existing character into the sketch, like in the following line of the code: `u8g2.drawUTF8(50, 20, "☂");`

The second is to create a *char* array, which has two values: the first value is a hexadecimal number of the *Unicode* character and the second value is a null character. This can be done by using the *char* array called *COPYRIGHT_SYMBOL*, like in the following lines of code:

```
const char COPYRIGHT_SYMBOL[] = {0xa9, '\0'};
```

```
u8g2.drawUTF8(95, 20, COPYRIGHT_SYMBOL); //COPYRIGHT SYMBOL
```

The third way of using the function is to use a hexadecimal number for the character itself, like in the following line of code:

```
u8g2.drawUTF8(115, 15, "\xb0"); // DEGREE SYMBOL
```

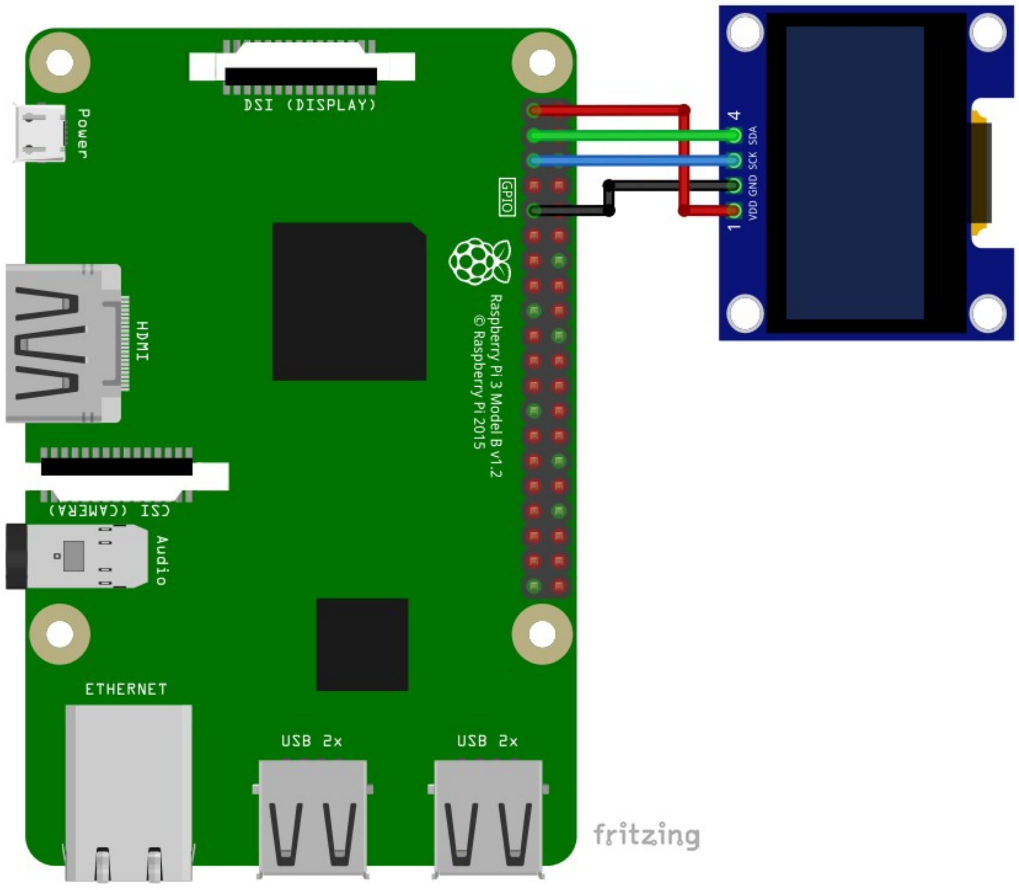
The function returns a value, an integer number which represents the width of the text (string).

To display something on the screen, the screen data buffer has to be cleared, then new value for data buffer has to be set (an image) and then to send the new value to the screen. This way, a new image will be displayed on the screen. In order to make this change visible, *`delay()`* function has to be used to delay the next change of the data buffer, like in the following lines of code:

```
u8g2.clearBuffer();
u8g2_bitmap(); // setting the data buffer
u8g2.sendBuffer();
delay(time_delay);
```

Connecting the screen with Raspberry Pi

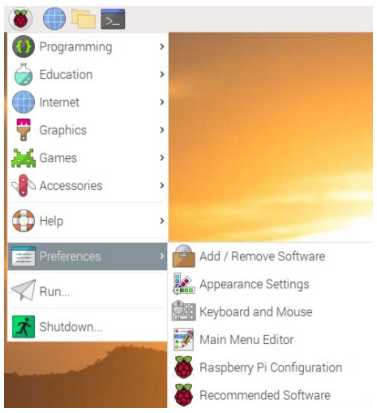
Connect the screen with the Raspberry Pi as shown on the following connection diagram:



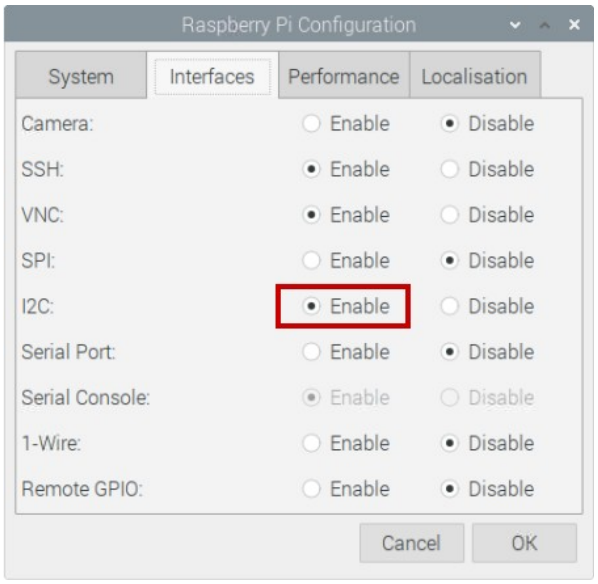
Screen pin	Raspberry Pi pin	Physical pin	Wire color
SDA	GPIO2	3	**Green wire**
SCL	GPIO3	5	**Blue wire**
GND	GND	9	**Black wire**
VCC	3V3	1	**Red wire**

Enabling the I2C interface

In order to use the module with Raspberry Pi, I2C interface has to be enabled. Open following menu:
Application Menu > Preferences > Raspberry Pi Configuration



In the new window, under the tab *Interfaces*, enable the I2C radio button, as on the following image:


Raspberry Pi Configuration

System | Interfaces | Performance | Localisation

Camera:	Enable	[✓ Disable]
SSH:	[✓ Enable]	Disable
VNC:	[✓ Enable]	Disable
SPI:	Enable	[✓ Disable]
I2C:	[✓ Enable]	Disable
Serial Port:	Enable	[✓ Disable]
Serial Console:	[✓ Enable]	Disable
1-Wire:	Enable	[✓ Disable]
Remote GPIO:	Enable	[✓ Disable]

Cancel OK

Libraries and tools for Python

To use a 1.3 inch OLED screen with a Raspberry Pi it is recommended to download an external library for it. The library that is to be used is called "*luma.oled*", but before that, few tools for Python has to be installed.

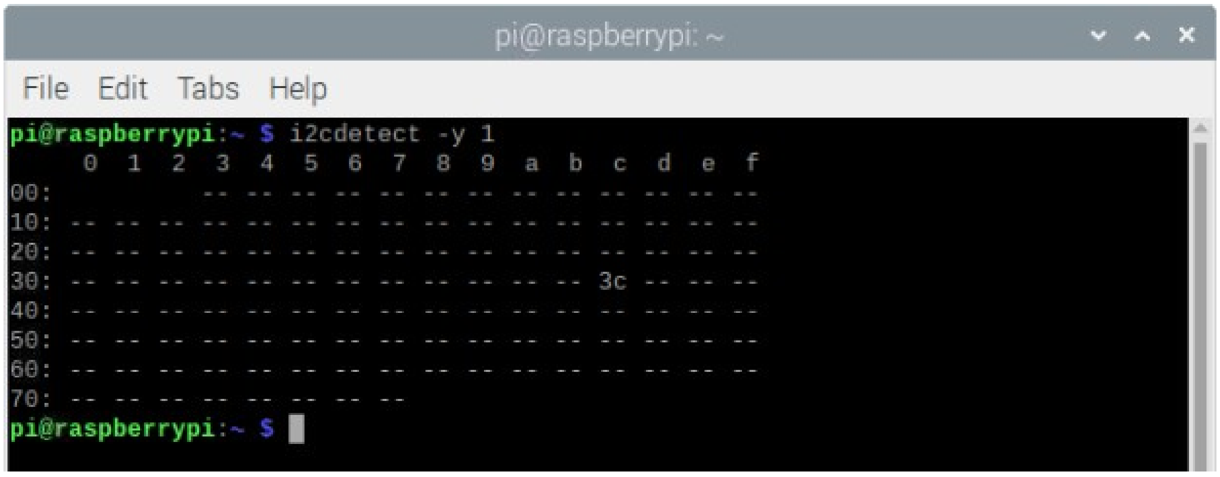
Open the terminal and run the following command:

```
sudo apt install i2c-tools python3-dev python3-pip libfreetype6-dev libjpeg-dev build-essential libopenjp2-7 libtiff5 git
```

Next, the screen I2C address has to be detected. In the terminal window, run the following command:

```
i2c detect -y 1
```

The result should be like the output on the following image:



```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- 3c -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi:~ $
```

where *0x3c* is the I2C address of the screen.

To install the *luma.oled* library, open the terminal and run the following command:

```
sudo -H pip3 install –upgrade luma.oled
```

The last thing to do is to download script examples from *GitHub*. In the terminal window run the following command:

```
git clone https://github.com/rm-hull/luma.examples.git
```

Then, (in terminal) change directory to *luma.examples* directory, with the following command: **`cd luma.examples`**

and to install it, run the following command:

```
sudo -H pip3 install -e .
```

(with the dot at the end)

Python script

The following script code is modified code from two scripts: */luma.examples/demo.py* and */luma.examples/demo_opts.py*.

```
import time
import sys
from luma.core.interface.serial import i2c
from luma.oled.device import sh1106
from luma.core.render import canvas
from PIL import ImageFont

font_path = 'ChiKareGo.ttf'

def changing_var(device):
    size = 40
    nf = ImageFont.truetype(font_path, size) # new font

    for i in range(100):
        with canvas(device) as draw:
            draw.text((28, 7), 'Changing var.', fill=1)
            if i < 10:
               draw.text((50,22),'0{}'.format(str(i)),font=nf,fill=1)
            else:
               draw.text((50, 22), str(i), font=nf, fill=1)

            time.sleep(0.001)
```

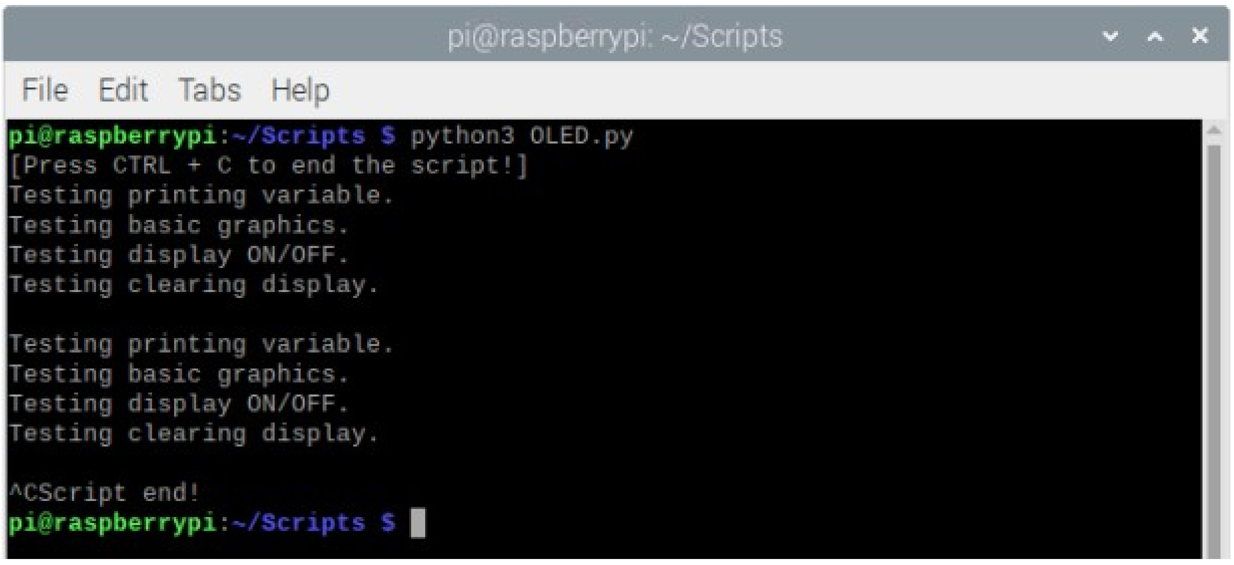
```
def primitives(device):
    with canvas(device) as draw:
        # Draw a rectangle.
        draw.rectangle((4, 4, 40, 10), outline=1, fill=0)

        # Draw an ellipse.
        draw.ellipse((4, 20, 18, 34), outline=1, fill=1)

        # Draw a triangle.
        draw.polygon([(10,44),(40,20),(40,44)],outline=1,fill=0)

        # Draw an X.
        draw.line((4, 48, 126, 62), fill=1)
        draw.line((4, 62, 126, 48), fill=1)

        # Write two lines of text.
        draw.text((45, 20), 'AZ-Delivery', fill=1)

        size = 10
        nf = ImageFont.truetype(font_path, size)
        draw.text((45, 4), 'AZ-Delivery', font=nf, fill=1)
```

```
try:
    serial = i2c(port=1, address=0x3c)
    device = sh1106(serial, rotate=0, width=128, height=64)

    print('[Press CTRL + C to end the script!]')
    while(True):
        print('Testing printing variable.')
        changing_var(device)
        time.sleep(2)

        print('Testing basic graphics.')
        primitives(device)
        time.sleep(3)

        print('Testing display ON/OFF.')
        for _ in range(5):
           time.sleep(0.5)
           device.hide()
           time.sleep(0.5)
           device.show()

        print('Testing clearing display.\n')
        device.clear()
        time.sleep(2)

except KeyboardInterrupt:
    print('Script end!')
```

Save the script under the name "*OLED.py*". To run the script, open terminal in the directory where script is saved and run the following command:

`python3 OLED.py`

The result should look like the output on the following image:



```
pi@raspberrypi:~/Scripts $ python3 OLED.py
[Press CTRL + C to end the script!]
Testing printing variable.
Testing basic graphics.
Testing display ON/OFF.
Testing clearing display.

Testing printing variable.
Testing basic graphics.
Testing display ON/OFF.
Testing clearing display.

^CScript end!
pi@raspberrypi:~/Scripts $ 
```

To stop the script press 'CTRL + C' on the keyboard.

The script starts with importing libraries and functions.

Then the path is saved to the special font in the variable *`font_path`*. The font used is called *`ChiKareGo.ttf`*, which can be found in: *luma.examples > examples > fonts > ChiKareGo.ttf*
The font file must be copied into the same directory where script is saved.

Then, two functions are created, the first is called *`changing_var()`* and the second is called *`primitives()`*.

The *`changing_var()`* function is used to display a variable which changes its state every millisecond. The function accepts one argument and returns no value. The argument is called a *`device`* argument, which will be explained later in the text. At the beginning of the *`changing_var()`* function, *`size`* variable is defined and initialized with the number *40*. This value represents the size of the font. Then, a variable called *`nf`* (*`new_font`*) is created, which represents the font that is used. The *`size`* and *`font_path`* variables are used to create the *`nf`* variable. Next, the *`for`* loop is created to loop through the first hundred integer numbers. These values are used as values for the variable that will be displayed on the screen. In the *`for`* loop, image is created which will be displayed on the screen, with the following line of the code:

```
with canvas(device) as draw:
```

where an object called a s*`draw`* is created. The *`draw`* object represents the image itself.

To output the text, the *text()* function is used. The *text()* function accepts four arguments, where one is optional, and returns no value. The first argument is a *tuple* with two elements, where the elements represent the *X* and *Y* positions of the cursor. The second argument represents the text itself, a *string* value. The third argument is the *fill* argument. The fourth argument is optional, it is called the *font* argument. If argument is not used, the font will be set to default font and size. But if *nf* variable is stored inside the *font* argument, desired font can be used and the font size changed.

The *fill* argument represents the color of the text. If the *fill = 0* the color will be black (nothing displayed on the screen) and if the *fill* is equal to any other value, for example *fill = 1*, the color of the text will be white (because OLED screens have only black and white colors). This library can also be used for many other screens, so inside the *fill* argument, any other color can be stored, which is not possible for a 1.3 inch OLED screen.

The rest of code inside the *for* loop is an algorithm to display text and variable with two digit numbers starting from *00* to *99*.

The second function is called *primitives()* and it is used to draw many shapes on the screen. It too, accepts one argument, the *device* argument, and returns no value. At the beginning of the function, the image object *draw* is created. This object is used to draw many shapes.

To draw the rectangle, the *`rectangle()`* function is used. The function accepts three arguments and returns no value. The first argument is a *`tuple`* of four elements, where the first two elements represent *X* and *Y* positions of the top right corner of the rectangle. The third and the fourth element represents the *X* and *Y* positions of the bottom right corner of the rectangle. The second argument is the *`outline`* argument and the third argument is the *`fill`* argument.

The *X* position value starts at the left side of the screen (value of *0*) and ends at the right side of the screen (value of *127*). The *Y* position value starts at the top side of the screen (value of *0*) and ends at the bottom side of the screen (value of *63*).

The *`outline`* argument represents the color of the shape edge and the *`fill`* argument represents the color of the shape itself. Because OLED screens are used, the colors are black and white, black - the pixel is turned *OFF*, white - the pixel is turned *ON*. When the value of zero is saved to the *`outline`* or *`fill`* argument this means that it is a black color. When any other value greater than zero is saved, for example *1*, this represents white color.

To draw ellipse or circles the *ellipse()* function is used. The function accepts three arguments and returns no value. The first argument is a *tuple* of four elements. First two elements represent the *X* and *Y* positions of the top left corner of the rectangle that contains the ellipse. Second two elements represent the *X* and *Y* positions of the bottom right corner of the rectangle that contains the ellipse. The second argument is the *outline* argument and the third argument is the *fill* argument.

To draw a polygon, the *polygon()* function is used. A polygon is a triangle, a rectangle, or any other shape with 3 or more corners. The function accepts three arguments. The first argument is a *list* of three or more *tuples*. The *tuples* have two elements, which represent the *X* and *Y* positions of the shape corner point. The number of *tuples* in the *list* is arbitrary, three or more *tuples*, which depends on what shape is wanted to be drawn. The second argument is the *outline* argument and the third argument is the *fill* argument.

To draw a line, the *line()* function is used. The function accepts two arguments and returns no value. The first argument is a *tuple* of four elements, where the first two elements represent *X* and *Y* positions of the starting point of a line and the second two elements represent *X* and *Y* positions of the end point of a line. The second argument is the *fill* argument.

At the end of the *primitives()* function, the *text()* function is used with and without *font* argument to show the difference.

After these two functions a*try-except* block of code is created. In the *try* block of code, two objects and the indefinite loop are created.

The first object is called *serial* and it is used to set up the I2C interface and address for the screen. To initialize this object, the following line of the code is used:

```
serial = i2c(port=1, address=0x3c)
```

The second object is called a *device*. This object represents the driver chip itself, and is used to control the driver chip. To initialize the device object the *serial* object, *width* and *height* arguments are used, in the following line of code:

```
device = sh1106(serial, width=128, height=64)
```

Next, an indefinite loop (*while True:*) is created. Inside the indefinite loop, two functions are excuted and two properties of the screen tested. First, the *changing_var()* function is executed, then the *primitives()* function. After this, *device.hide()* is used to turn *OFF* the screen and *device.show()* to turn it *ON* again. At the end of the indefinite loop, *device.clear()* is used to clear the data buffer of the screen, which clears the screen image too.

The *`except`* block of code is executed when 'CTRL+C' is pressed on the keyboard.
This is called the keyboard interrupt. When the *`except`* block of code is executed, the message “*`Script end!`*” is printed in the terminal.

Now it is the time to learn and make your own projects. You can do that with the help of many example scripts and other tutorials, which can be found on the Internet.

If you are looking for the high quality products for Arduino and Raspberry Pi, AZ-Delivery Vertriebs GmbH is the right company to get them from. You will be provided with numerous application examples, full installation guides, eBooks, libraries and assistance from our technical experts.

https://az-delivery.de

Have Fun!

Impressum

https://az-delivery.de/pages/about-us