

AZ-Delivery

Welcome!

Thank you for purchasing our *AZ-Delivery 1.77 inch TFT LCD SPI screen*.
On the following pages, we will introduce you to how to use and set-up this handy device.

Have fun!



Areas of application

Education and teaching: Use in schools, universities and training institutions to teach the basics of electronics, programming and embedded systems. Research and development: Use in research and development projects to create prototypes and experiments in the fields of electronics and computer science. Prototype development: Use in the development and testing of new electronic circuits and devices. Hobby and Maker Projects: Used by electronics enthusiasts and hobbyists to develop and implement DIY projects.

Required knowledge and skills

Basic understanding of electronics and electrical engineering. Knowledge of programming, especially in the C/C++ programming language. Ability to read schematics and design simple circuits. Experience working with electronic components and soldering.

Operating conditions

The product may only be operated with the voltages specified in the data sheet to avoid damage. A stabilized DC power source is required for operation. When connecting to other electronic components and circuits, the maximum current and voltage limits must be observed to avoid overloads and damage.

Environmental conditions

The product should be used in a clean, dry environment to avoid damage caused by moisture or dust. Protect the product from direct sunlight (UV), as this can negatively affect the lifespan of the display.

Intended Use

The product is designed for use in educational, research and development environments. It is used to develop, program and prototype electronic projects and applications. The product is not intended as a finished consumer product, but rather as a tool for technically savvy users, including engineers, developers, researchers and students.

Improper foreseeable use

The product is not suitable for industrial use or safety-relevant applications. Use of the product in medical devices or for aviation and space travel purposes is not permitted.

disposal

Do not discard with household waste! Your product is according to the European one Directive on waste electrical and electronic equipment to be disposed of in an environmentally friendly manner. The valuable raw materials contained therein can be recycled become. The application of this directive contributes to environmental and health protection. Use the collection point set up by your municipality to return and Recycling of old electrical and electronic devices. WEEE Reg. No.: DE 62624346

electrostatic discharge

The display is sensitive to electrostatic discharge (ESD), which can damage or destroy the electronic components. Please note the following safety instructions to avoid ESD hazards: Attention: Electrostatic charges on your body can damage the display. Note: Ground yourself by wearing an anti-static wrist strap connected to a grounded surface or by touching a grounded metal surface before handling the display. Attention: Use anti-static mats and bags to protect the display. Note: Place the display on an anti-static work mat and store in anti-static bags when not in use. Note: A clean and grounded workplace minimizes the risk of ESD. Action: Keep your workplace clean and free of materials that can generate electrostatic charges. Make sure all surfaces used are grounded.

safety instructions

Although the display complies with the requirements of the RoHS Directive (2011/65/EU) and does not contain any hazardous substances in quantities above the permitted limits, residual chemical hazards may still exist. Please note the following safety instructions: Attention: The back of the display and the circuit board can release chemical residues from manufacturing or during operation. Note: Wear protective gloves when handling or installing the display for a long time to avoid skin irritation. Caution: Electronic components can emit small amounts of volatile organic compounds (VOCs), especially if the display is new. Note: Make sure you work in a well-ventilated area to minimize the concentration of fumes in the air. Caution: Do not use harsh chemicals or solvents to clean the display as they may damage the protective coating or electronics. Note: Use an anti-static cleaning cloth or special electronics cleaner to carefully clean the display. Although the display complies with

the requirements of the RoHS Directive (2011/65/EU) and does not contain any hazardous substances in quantities above the permitted limits, residual chemical hazards may still exist. Please note the following safety instructions: Attention: The back of the display and the circuit board can release chemical residues from manufacturing or during operation. Note: Wear protective gloves when handling or installing the display for a long time to avoid skin irritation. Caution: Electronic components can emit small amounts of volatile organic compounds (VOCs), especially if the display is new. Note: Make sure you work in a well-ventilated area to minimize the concentration of fumes in the air. Caution: Do not use harsh chemicals or solvents to clean the display as they may damage the protective coating or electronics. Note: Use an anti-static cleaning cloth or special electronics cleaner to carefully clean the display. The display contains sensitive electronic components and a top layer. Improper handling or excessive pressure can cause damage to the display or injury. Observe the following safety instructions to avoid mechanical hazards: Attention: The cover of the display is fragile and can break if handled improperly. Note: Avoid applying strong pressure or bending the display. Handle the display carefully and only by the circuit board to avoid breakages. Caution: Drops or impacts can crack the surface of the display and damage the electronic components on the back. Note: Avoid dropping the display and protect it from impacts. Use a soft surface when working to avoid scratches. Attention: If the display breaks, sharp pieces of glass can cause injuries. Note: If the display breaks, handle the fragments carefully and wear protective gloves to avoid cuts. Dispose of the glass pieces safely. Note: Improper attachment can lead to mechanical stress and breakage of the display. Action: Attach the display securely and without excessive pressure. Use appropriate brackets or housings to mount the display stably. Caution: Improper cleaning methods may scratch or damage the surface. Note: Only use soft, anti-static cloths to clean the display. Avoid aggressive cleaning agents and strong friction. The display operates with electrical voltages and currents that, if used improperly, can cause electric shocks, short circuits or fires. Please note the following safety instructions: Attention: Use the product only with the specified voltages. Note: The performance limits of the product can be found in the associated data sheet. Note: Improper voltage sources can damage the display or cause dangerous situations. Action: Only use tested and suitable power supplies or batteries to power your circuits. Make sure the voltage source meets the requirements of the display. Caution: Avoid short circuits between the connectors and components of the product. Note: Make sure that no conductive objects touch or bridge the circuit board. Use insulated tools and pay attention to the arrangement of connections. Caution: Do not perform any work on the product when it is connected to a power source. Note: Disconnect the product from power before making any circuit changes or connecting or removing components. Note: Look for signs of electrical damage such as smoke, unusual odors, or discoloration. Action: If such signs occur, turn off the power immediately and inspect the circuit thoroughly for errors. The display can generate heat during operation, which could lead to overheating, burns or fire if handled improperly. Please note the following safety instructions: Attention: Some components of the display can heat up during operation or in the event of an error. Measure: After switching off, allow the display to cool down sufficiently before touching the individual components on the back directly. Avoid direct contact with hot components. Caution: Overloading can cause excessive heating of the electronic components. Note: Make sure the power and voltage supply meets the specifications of the display and does not cause overload.

Az-Delivery

The Liquid Crystal Display, or LCD for short, is a device that uses liquid crystals to block light. Liquid crystals do not emit light directly, but block light coming from the backlight of the screen therefore producing shapes on the screen. If you are using a monochrome LCD screen, liquid crystals are just shadows on the screen. But if you are using a colour LCD screen, liquid crystals are colour filters which block only parts of white light that comes from the backlight of a screen, thus producing specific colour.

Thin Film Transistor Liquid Crystal Display, or TFT LCD for short, is a variant of a LCD screens with Thin Film Transistor technology. We will not go into details, but TFT decreases the size of the specific liquid crystal filter, which improves readability and addressability of each pixel of the screen (this improves contrast of the screen).

At rest, liquid crystals allow background light to pass through them. The crystals are in the chaotic state. But if you connect power to some part of the screen, crystals align themselves and create shade or filter for the background light.

The 1.77 inch TFT LCD screen can produce 65536 different colours. It has 128x160 pixels, and the driver chip of the screen is the IC driver called "ST7735". To communicate with the driver chip we have to use Serial Peripheral Interface, or SPI for short.



Specifications

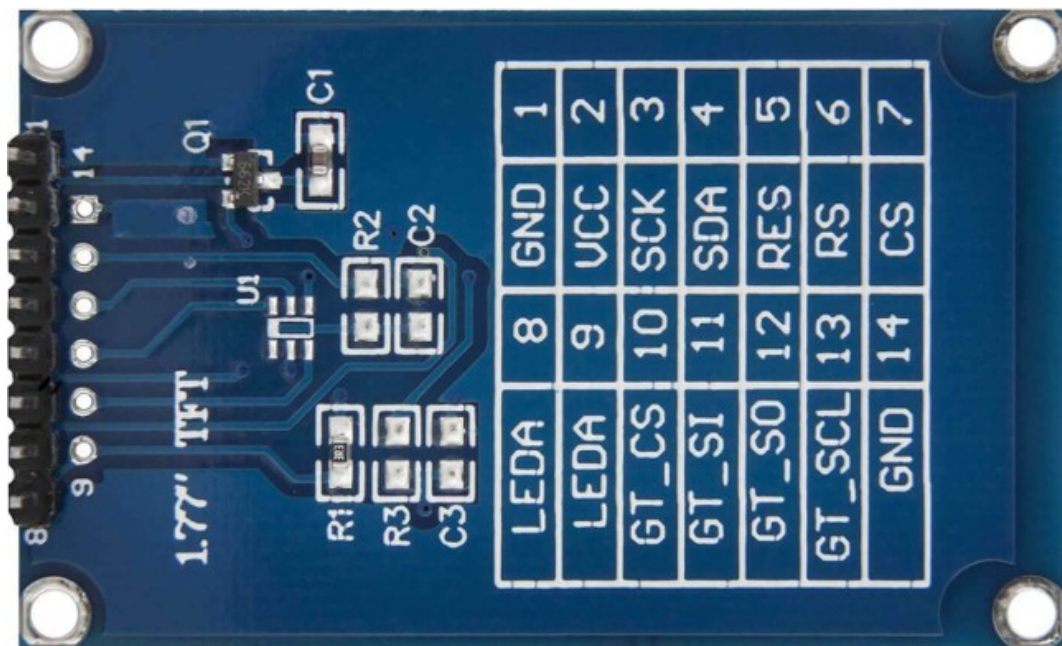
- » Operating voltage: from 2.7V to 3.3V DC
- » Operating current: ~50mA
- » Screen type: LCD TFT
- » Screen driver IC: ST7735
- » Driver IC interface: SPI interface
- » Screen size: 1.77 inch diagonal
- » Screen resolution: 128x160 pixels
- » Screen colours: 16 bit colours or 65536 colours
- » Dimensions (display): 32 x 38mm [1.26 x 1.50in]
- » Dimensions (PCB): 34 x 56mm [1.34 x 2.20in]

Many libraries, that are used for these screens, have examples that show you how to draw lines, rectangles, triangles, circles or formatted text on the screen. These examples often use ESP32 or ESP8266 Micro Controllers. The *ATMega328* microcontroller of the Atmega328P Board does not have enough memory to work with high speed graphics. Nevertheless, our simple example will be based on the Atmega328P Board.

Az-Delivery

The pinout

LEDA
CS
RS
RES
SDA
SCK
VCC
GND



Note: Pins 9 to 14 are not used

Az-Delivery

Connect the *LEDA* pin to the +3.3V to turn *ON* the screen, or you can use a potentiometer to adjust the level of brightness. You can even use the digital pin of the microcontroller with PWM to adjust the level of brightness. The background light requires less than one milliamp.

Warning: Do not connect *LEDA* pin to +5V, it could destroy your screen!

Connect the *VCC* pin to +5V and *GND* pin to the ground in order to power up the screen. For this purpose there is on-board +3.3V voltage regulator (type: 662K).

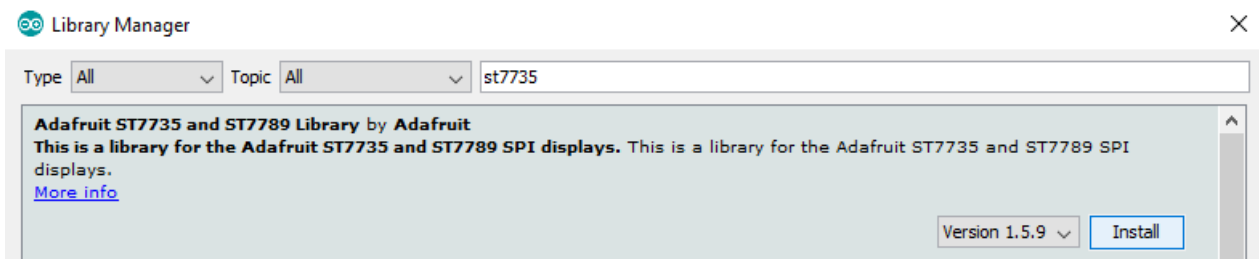
Other pins are used to control and to communicate with the screen.

Warning: According to the data sheet logic pins work only on +3.3V! We strongly recommend a logic level converter (LLC) for 5V MCUs (see below). Nevertheless, we also connected the SPI pins directly to the Atmega328P Board and it worked.

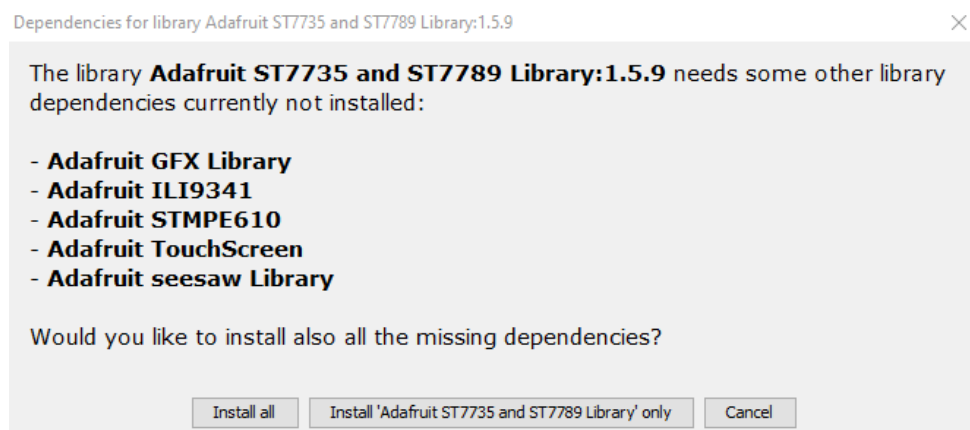
Az-Delivery

1.77 inch screen and Atmega328P Board

To use the 1.77 TFT LCD SPI screen with the Atmega328P Board, it is recommended to download and install an external library. The library we are going to use is called the “*Adafruit_ST7735*”. To download it, open Arduino IDE and go to: *Tools > Manage Libraries*. New window will open, type “*ST7735*” in the search box and install a library called “*Adafruit_ST7735 and ST7789 Library*” made by “*Adafruit*” as shown on the following image:

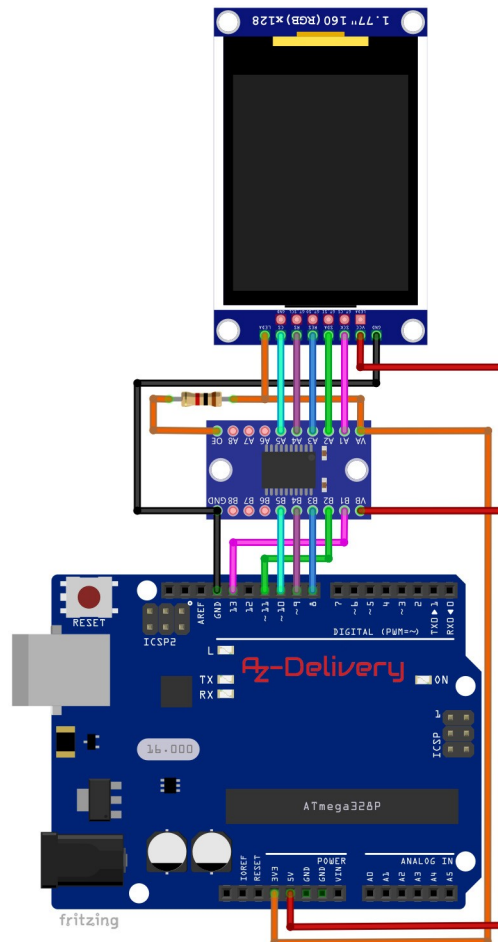


When you click on the “*Install*” button, a new window will open, requesting from you to install dependencies, libraries that are used with “*Adafruit_ST7735*” library. Click “*Install all*” button and wait for a few moments for everything to be installed. The request window can be seen on the following image:



Az-Delivery

Connect the 1.77 inch TFT LCD SPI screen with the Atmega328P Board as shown on the following connection diagram:



TFT LCD pin	>	LLC pin	>	Pin
SCK	>	A1 - B1	>	D13
SDA (MOSI)	>	A2 - B2	>	D11
RES (RESET)	>	A3 - B3	>	D8
RS (REG. SEL.)	>	A4 - B4	>	D9
CS (SS)	>	A5 - B5	>	D10
LEDA	>	VA	>	3.3V
VCC	>	VB	>	5V
GND	>	GND	>	GND
	>	OE (via 1kΩ)	>	3.3V

Pink wire

Green wire

Blue wire

Purple wire

Cyan wire

Orange wire

Red wire

Black wire

Orange wire

AZ-Delivery

NOTE: Logic pins of the 1.77 inch screen are not 5V tolerant, so you have to use a device called a logic level converter (or shifter), or the LLC for short. AZ-Delivery offers a device called a “*TXS0108E Logic level converter*”. To set-up this device you have to do a few things. First, you have to connect *VA* pin of the LLC to the +3.3V. Next connect *VB* pin of the LLC to +5V. Then you have to connect *OE* pin to +3.3V via a 1kΩ pull up resistor. Lastly, connect *GND* pin of the LLC to the ground. “A” side of the LLC is used for low level voltages (3.3V), and “B” side of the LLC is used for high level voltages (5V), as you can see on the connection diagram. You can read more about this device in our free eBook “*Quick Starter Guide for TXS0108E Logic level converter*” which can be found on our site:

https://www.az-delivery.de/products/logiklevel-wandler-kostenfreies-e-book?_pos=1&_sid=06ac64d83&_ss=r



Sketch example

With the installation of the Adafruit library, a couple of examples are saved in the microcontroller directory and can be found under File/Examples/Adafruit ST7735 and ST7789. As mentioned above, these files require an ESP32 or ESP8266 Micro Controller. Therefore, we modified the sketch ***graphicstest*** for the microcontroller board with Atmega328P.

```
#include <Adafruit_GFX.h>    // Core graphics library
#include <Fonts/FreeSansBold9pt7b.h>
#include <Adafruit_ST7735.h> // Hardware-specific library
#include <SPI.h>
```

```
Adafruit_ST7735 tft = Adafruit_ST7735(10, 8, 9);
```

```
void setup(void) {
  tft.initR(INITR_GREENTAB);
  tft.fillScreen(ST77XX_BLACK);
  delay(500);

  show_page();
  tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
  tft.setTextSize(2);
  tft.setFont(); // reset font to default
}
```

```
void loop() {
  for(uint8_t i = 0; i < 100; i++) {
    changing_value(i);
    // delay(100);
  }
}
```

AZ-Delivery

```
void show_page() {
    tft.setFont(&FreeSansBold9pt7b);
    tft.fillScreen(ST77XX_BLACK);
    tft.setTextColor(ST77XX_RED);
    tft.setCursor(14, 22);
    tft.print("AZ-Delivery");

    tft.drawFastHLine(0, 35, 128, ST77XX_GREEN);

    tft.drawTriangle(1, 45, 28, 70, 55, 45, ST77XX_WHITE);
    tft.fillTriangle(78, 70, 104, 45, 127, 70, 0xA3F6);

    tft.drawRect(1, 80, 50, 30, ST77XX_BLUE);
    tft.fillRoundRect(78, 80, 50, 30, 5, 0x2D4E);

    tft.fillCircle(25, 135, 15, 0x5BA9);
    tft.drawCircle(102, 135, 15, ST77XX_GREEN);

    tft.drawLine(45, 150, 80, 40, ST77XX_ORANGE);
}

void changing_value(uint8_t value) {
    if(value < 10) {
        tft.setCursor(15, 88);
        tft.print("0");
        tft.print(value);
    }
    else {
        tft.setCursor(15, 88);
        tft.print(value);
    }
}
```

Az-Delivery

Sketch starts with importing of four libraries. First library is the core graphics library, second is for a specific font, third is the hardware specific library (for the driver IC ST7735), and fourth is a library for the SPI communication.

We create an object called “*tft*” with the following line of the code:

```
Adafruit_ST7735 tft = Adafruit_ST7735(10, 9, 8);
```

where, *10* represents digital I/O pin of the Atmega328P Board on which *CS* pin of the screen is connected; *9* represents digital I/O pin of the microcontroller board on which *RS* pin of the screen is connected; and, *8* represents digital I/O pin of the Atmega328P Board on which *RES* pin of the screen is connected.

tft object is used to set-up settings of the screen or to send data to the screen.

In the *setup()* function we initialize *tft* object with the following line of the code: `tft.initR(INITR_GREENTAB);`

where *INITR_GREENTAB* represents special initializer for the 1.77 inch TFT LCD SPI screen. The libraries can be used with the other LCD screens too, that is why there are different initializers.

After this, we use built-in function to fill the whole screen with the specific predefined colour: `tft.fillScreen(ST77XX_BLACK);`

where *ST77XX_BLACK* represents predefined black colour.

Az-Delivery

You can use any of the following predefined colours:

ST77XX_BLACK

ST77XX_WHITE

ST77XX_RED

ST77XX_GREEN

ST77XX_BLUE

ST77XX_CYAN

ST77XX_MAGENTA

ST77XX_YELLOW

ST77XX_ORANGE

Or, you can use 4-digit hex numbers that represent specific colour (for example `0x2AFF`).

Then, we execute function called “*show_page()*” which will be explained later in the text.

At the end of the *setup()* function we use three functions that are used to set-up settings for printing text on the screen.

First function is called a “*setTextColor()*” which accepts two arguments and returns no value. The second argument is optional. Values of both arguments are hex numbers for colours, where first argument is for text colour and the second is for background colour around the text. Only if you are not using default font for the text, the second argument will be ignored.

Second function is called a “*setTextSize()*” which accepts one argument and returns no value. The argument value is an integer number in the range from 1 to 5. The integer number represents the size of the text, where 1 is the smallest text size and 5 is the largest text size.

Az-Delivery

And, the third function is called a “*setFont()*” which can accept no argument, or one optional argument, and returns no value. If the *setFont()* function is used without argument, this resets the font of the text to the default font. But, when you pass the optional argument to the *setFont()* function, you can change the font of the text. The value of the optional argument is the name of the font, for example:

```
setFont(&FreeSansBold9pt7b)
```

The fonts can be found in the *Adafruit_GFX* library folder:

... > *Arduino* > *libraries* > *Adafruit_GFX_Library* > *Fonts*

When you are importing specific font you have to import the font library after *Adafruit_GFX* library, like in the following lines of the code:

```
#include <Adafruit_GFX.h>  
#include <Fonts/FreeSansBold9pt7b.h>
```

All fonts from the *Adafruit_GFX* library can be found on the last page of this eBook.

Az-Delivery

In the `loop()` function we use *for* loop to change the value of the variable which will be printed on the screen. In every loop of the *for* loop, we execute the `changing_value()` function which will be explained later in the text.

In the `show_page()` function we use several built-in functions of the `Adafruit_ST7735` library. At the beginning of the `show_page()` function we use three functions that we already wrote about, `setFont()`, `setTextColor()` and `fillScreen()`.

The `setCursor()` function accepts two arguments and returns no value. The first argument is *X* position of the cursor (in range from 0 to 127), where *X* position starts in top left corner of the screen and ends in the top right corner of the screen. The second argument is *Y* position of the cursor (in range from 0 to 159), where *Y* position starts in top left corner of the screen and ends in the bottom left corner of the screen. You have to use `setCursor()` function before `print()` function in order to point the `print()` function where to print the text. If you do not use `setCursor()` function before `print()` function, the `print()` function prints text on the (0, 0) position of the cursor, or on the last known cursor position.

The `print()` function accepts two arguments, where the second one is optional, and returns no value. First argument value is the text which will be printed on the screen. Text can be a string, integer number, float/double number or character.

Az-Delivery

If you are using the `print()` function with the second argument, then first argument has to be an integer number and second argument value has to be one of these: `DEC`, `OCT` or `HEX`. This option allows you to print an integer number in one of the following number systems: `DEC` = decimal, `OCT` = octal and `HEX` = hexadecimal number system.

There is a version of `print()` function which is called a `println()` function. The difference is that the `println()` function prints text and sets the cursor to a new line.

The `drawFastHLine()` function accepts four arguments and returns no value. The function is used to draw a horizontal line. First two arguments are the `X` and `Y` position of the starting point of the line, third argument is the length of the line and fourth argument is the colour of the line.

The `drawTriangle()` function accepts seven arguments and returns no value. The function is used to draw an empty triangle. First two arguments are the `X` and `Y` position of the first point of the triangle, next two arguments are the `X` and `Y` position of the second point of the triangle, next two arguments are the `X` and `Y` position of the third point of the triangle and the last argument is the colour of a triangle.

The `fillTriangle()` function does the same as `drawTriangle()`, the difference is that `fillTriangle()` function draws a triangle that is filled with a colour.

Az-Delivery

The `drawRect()` function accepts five arguments and returns no value. The function is used to draw an empty rectangle. First two arguments are the *X* and *Y* position of the top left corner point of the rectangle, next two arguments are the *X* and *Y* position of the bottom right corner point of the rectangle and last argument is the colour of a rectangle.

The `fillRect()` function does the same as `drawRect()`, the difference is that `fillRect()` function draws a rectangle that is filled with a colour.

The `drawRoundRect()` function accepts six arguments and returns no value. The function is used to draw an empty rectangle with round corners. First two arguments are the *X* and *Y* position of the top left corner point of the rectangle, next two arguments are the *X* and *Y* position of the bottom right corner point of the rectangle, fifth argument is the corner radius and last argument is the colour of a rectangle.

The `fillRoundRect()` function does the same as `drawRoundRect()`, the difference is that `fillRoundRect()` function draws a rectangle with round corners that is filled with a colour.

The `drawCircle()` function accepts four arguments and returns no value. The function is used to draw an empty circle. First two arguments are the *X* and *Y* position of the center point of the circle, third argument is the circle radius and last argument is the colour of a circle.

Az-Delivery

The `fillCircle()` function does the same as `drawCircle()`, the difference is that `fillCircle()` function draws a circle that is filled with colour.

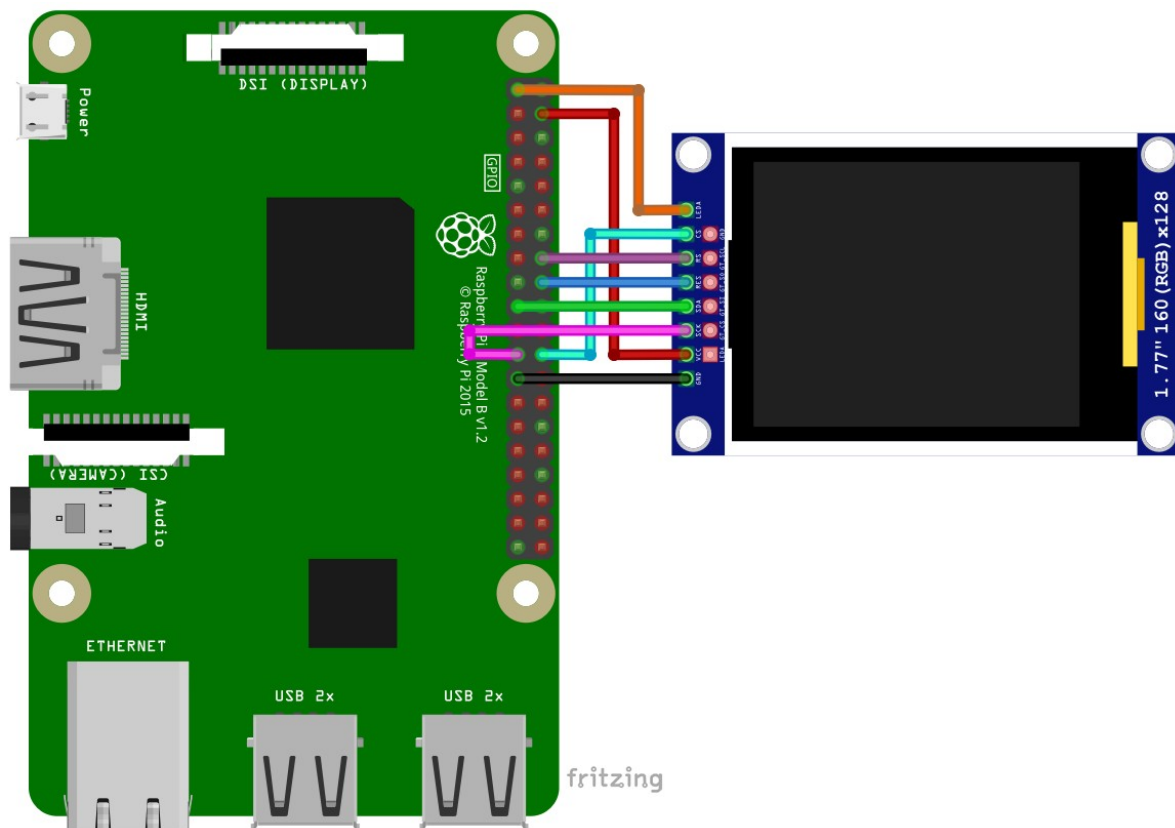
The `drawLine()` function accepts five arguments and returns no value. The function is used to draw a line. First two arguments are the *X* and *Y* position of the starting point of the line, next two arguments are the *X* and *Y* position of the ending point of the line and the last argument is for the colour of a line.

The `changing_value()` function accepts one argument and returns no value. The function is used to print variable which value is integer number on the specific location on the screen. The function checks if the integer number is less than 10. If it is, the function prints leading zero and number together, but if it is not, it just prints the number. The location where number will be printed is set-up with `setCursor()` function to the position (15, 88).

Because we are using default font, and we passed two arguments to `setTextColor()` function, the background colour of the text is black, so we do not need to worry about text printing itself on the last text value (which can make white square of the text). But if you are using any other font, you need to worry about this. There is a trick on how to clear the part of the screen where changing text is. First, print the text, then wait for a millisecond and then draw a black rectangle over the text (this clears the area where text is), and then print a new text over this.

Connecting the screen with Raspberry Pi

Connect the 1.77 inch TFT LCD SPI screen with a Raspberry Pi as shown on the following connection diagram:



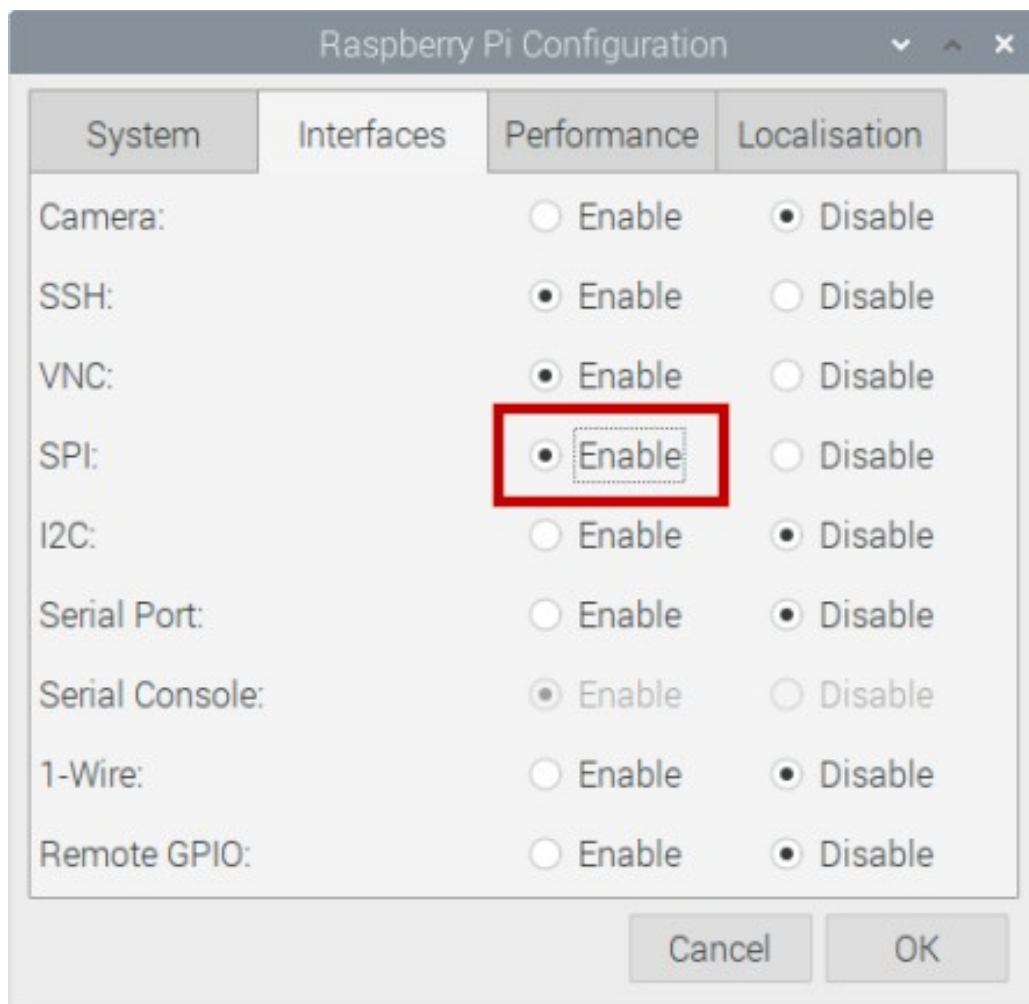
TFT LCD pin	>	Raspberry Pi pin	
LEDA	>	3.3V [pin 1]	Orange wire
CS (SS)	>	GPIO8/CE0 [pin 24]	Cyan wire
RS (REG. SEL.)	>	GPIO23 [pin 16]	Purple wire
RES (RESET)	>	GPIO24 [pin 18]	Blue wire
SDA (MOSI)	>	GPIO10/MOSI [pin 19]	Green wire
SCK	>	GPIO11/SCLK [pin 23]	Pink wire
VCC	>	5V [pin 4]	Red wire
GND	>	GND [pin 25]	Black wire

Enabling SPI interface

Before we can use the 1.77 inch TFT LCD SPI screen with a Raspberry Pi, first you have to enable the SPI interface in the Raspberry Pi OS (operating system, previously known as Raspbian). To do so, go to:

Application Menu > Preferences > Raspberry Pi Configuration

A new window will open. Go to “*Interfaces*” tab, find “*SPI:*” then check “*Enable*” radio button and click “*OK*” button, like on the following image:





Installing the library for Python

To use the 1.77 inch TFT LCD SPI screen with a Raspberry Pi it is recommended to download an external library. The library we are going to use is called "*luma.examples*". Before download and installation of the library it is recommended to update the operating system:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

To download a library run the following command in the terminal:

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

Then, change working directory to *luma.examples* directory, by running the following command in the terminal:

```
cd luma.examples
```

And to install the library run the following command (with dot at the end!):

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

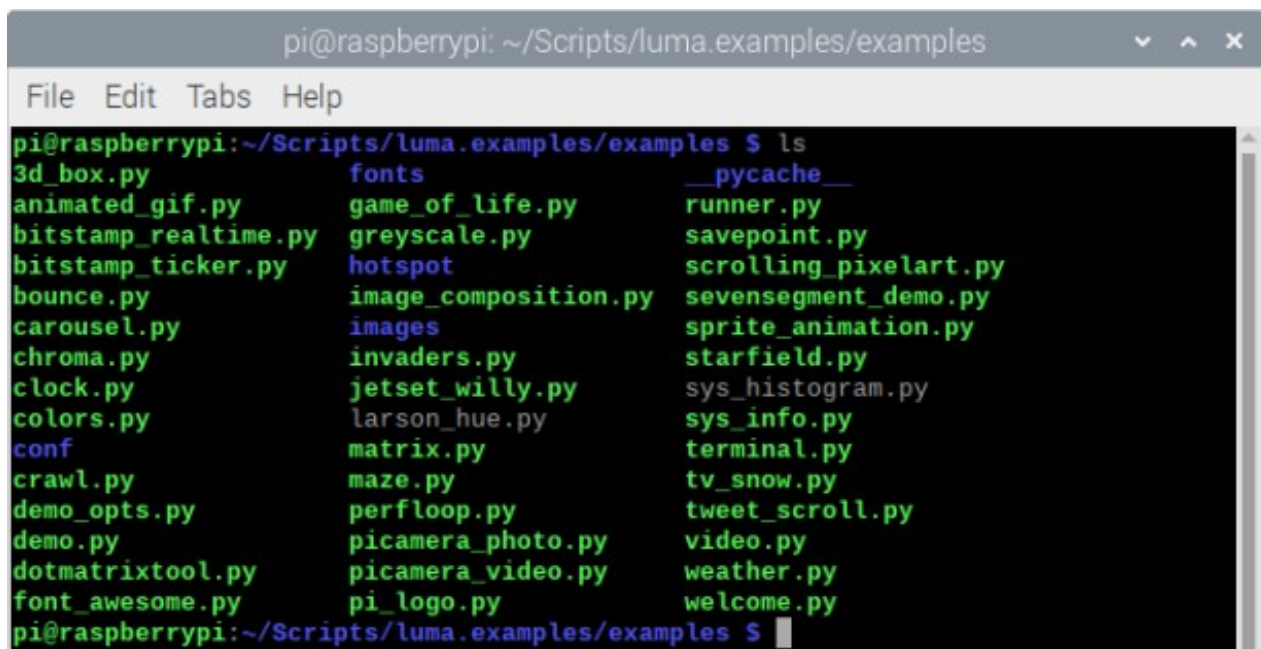
Az-Delivery

Running the examples

Open terminal in `luma.examples/examples` directory, then list all examples that come with the library by running the following command:

`ls`

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



```
pi@raspberrypi: ~/Scripts/luma.examples/examples
File Edit Tabs Help
pi@raspberrypi:~/Scripts/luma.examples/examples $ ls
3d_box.py          fonts              __pycache__
animated_gif.py    game_of_life.py   runner.py
bitstamp_realtime.py greyscale.py       savepoint.py
bitstamp_ticker.py hotspot           scrolling_pixelart.py
bounce.py          image_composition.py sevensegment_demo.py
carousel.py        images            sprite_animation.py
chroma.py          invaders.py        starfield.py
clock.py           jetset_willy.py   sys_histogram.py
colors.py          larsen_hue.py     sys_info.py
conf              matrix.py          terminal.py
crawl.py           maze.py            tv_snow.py
demo_opts.py       perfloop.py        tweet_scroll.py
demo.py            picamera_photo.py  video.py
dotmatrixtool.py   picamera_video.py  weather.py
font_awesome.py    pi_logo.py         welcome.py
pi@raspberrypi:~/Scripts/luma.examples/examples $
```

The luma library can be used for several displays, which have to be defined before the program may run with your specific display.

All above mentioned Python examples are predefined for the SSD1306 display with an I2C address of 0x3C. Therefore, they will not work with our 1,77" TFT with the ST7735 controller and SPI interface. They have to be adapted to our display. Or we use our own example program.

AZ-Delivery

Script example

```
# If not yet done please install the luma library by
# git clone https://github.com/rm-hull/luma.examples.git
# cd luma.examples

# sudo -H pip3 install -e .      (mind the dot at the end)

import time
import sys
from luma.core.interface.serial import spi
from luma.lcd.device import st7735
from luma.core.render import canvas
from PIL import ImageFont

font_path = '/home/luma.examples/examples/fonts/ChiKareGo.ttf'
s = spi(port=0, device=0, cs_high=True, gpio_DC=23, gpio_RST=24)
device=st7735(s, rotate=0, width=160, height=128, h_offset=0, v_offset=0, bgr=False)

def primitives(device, draw):
    # Draw a rectangle of the same size of screen
    draw.rectangle(device.bounding_box, outline='white')
    # Draw a rectangle
    draw.rectangle((4, 4, 80, 124), outline='blue', fill=(22, 55, 55))
    # Draw an ellipse
    draw.ellipse((6, 6, 78, 122), outline=(254, 155, 0), fill='green')
    # Draw a triangle
    draw.polygon([(90,124), (100,4), (120,124)], outline='blue', fill='red')
    # Draw an X
    draw.line((130, 4, 155, 124), fill='yellow')
    draw.line((130, 124, 155, 4), fill='yellow')
    # Print 'AZ-Delivery'
    draw.text((10, 60), 'AZ-Delivery', fill='red')
    # Change font adn size of text and print 'AZ-Delivery'
    size = 22
    new_font = ImageFont.truetype(font_path, size)
    draw.text((10, 70), 'AZ-Delivery', font=new_font, fill='red')

def changing_var(device):
    size = 40
    new_font = ImageFont.truetype(font_path, size)
```


Az-Delivery

```
for i in range(100):
    with canvas(device) as draw:
        draw.text((40, 38), 'Changing var.', fill='red')
        if i < 10:
            draw.text((63, 55), '0{}'.format(str(i)), font=new_font, fill='red')
        else:
            draw.text((63, 55), str(i), font=new_font, fill='red')
        time.sleep(0.02)

print('[Press CTRL + C to end the script!]\n')
try:
    while True:
        device.backlight(False)    # if backlight connected to GPIO18
        print('Testing printing variable.\n')
        changing_var(device)
        time.sleep(1)

        print('Testing basic graphics.\n')
        with canvas(device) as draw:
            primitives(device, draw)
        time.sleep(3)

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

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

except KeyboardInterrupt:
    device.backlight(True)    # if backlight connected to GPIO18
    device.cleanup()
    print('Script end!')
```

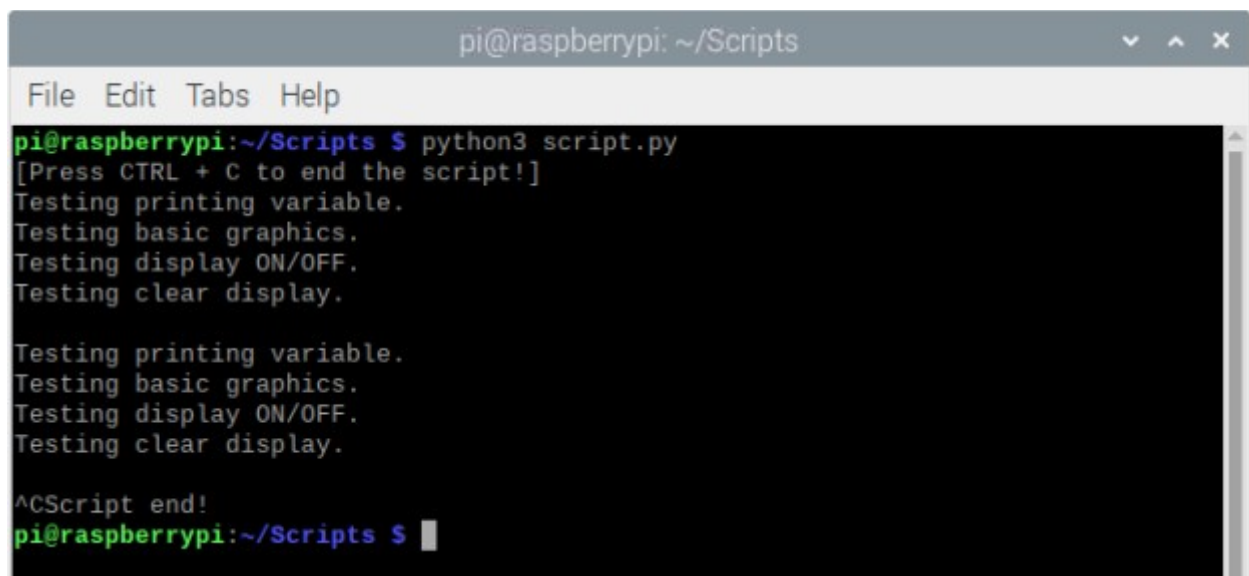
Az-Delivery

Save the script with the name “*script.py*”

To run the script, open terminal in the directory where you saved the script and run the following command:

python3 script.py

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

A screenshot of a terminal window titled "pi@raspberrypi: ~/Scripts". The window has a menu bar with "File", "Edit", "Tabs", and "Help". The terminal shows the command "python3 script.py" being executed. The output of the script is as follows:

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

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

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

To end the script press “*CTRL + C*” on the keyboard.

Az-Delivery

The script starts with importing libraries and functions.

After imports we create three objects: *font_path*, *s* and *device*.

The *font_path* object is used to specify the path to the font file.

The *s* object is used to create a serial interface for communication between the Raspberry Pi and driver IC of the screen. This object is created using a constructor called *spi()* from the *luma.core.interface.serial* library. This constructor accepts several keyword arguments and pre-defined arguments. Arguments can be *port* and *device* number, other arguments the GPIO assignment for CD and RST. In our script example we used: *port=0*, which is the SPI port of the Raspberry Pi; *device=0* because we are sending data via SPI interface to only one device named "0"; *gpio_DC=23*, a pin name of the Raspberry Pi on which *DC* (RS) pin of the screen is connected; and, *gpio_RST=24*, a pin name of the Raspberry Pi on which Reset (*RES*) pin of the screen is connected.

Az-Delivery

The *device* object is used to set-up settings for driver IC of the screen. This object is created using a constructor called *st7735()* from the *luma.lcd.device* library. The constructor accepts seven arguments. The first argument is the object for a serial communication (*s* object); the second argument is the *rotate* argument; the third argument is *width* argument; the fourth argument is the *height* argument; the fifth argument is the *h_offset* argument; the sixth argument is the *v_offset* argument; and, the seventh argument is the *bgr* argument. In our example script we used: *rotate=0*, the value of this argument is an integer number in range from 0 to 3, where 0 = 0°, 1 = 90°, 2 = 180° and 3 = 270°; *width=160*, the value of this argument has to be 160, because the width of our screen is 160 pixels; *height=128*, the value of this argument has to be 128, because the height of our screen is 128 pixels; *h_offset=0*, this is horizontal offset of the image and the value of this argument is an integer number; *v_offset=0*, this is vertical offset of the image and the value of this argument is an integer number; and, *bgr=False*, use the value "True" for this argument if you experience inverted colours on the screen. The values of the *width* and *height* argument should not be changed, because if you do change them you get the error when you start the script. The values of *h_offset* and *v_offset* are integer values, which should be used to move the image on the screen if you experience the white or green lines near any of the edges of the screen.

Az-Delivery

The self-defined function `primitives()` accepts two arguments and returns no value. The first argument is an object called a “*device*” used to indicate screen driver IC and how to set it up. The second argument is the “*draw*” object which is used for sending commands to the screen. Inside the `primitives()` function we use predefined functions from the `luma.core` library.

First predefined function is called a `rectangle()` which accepts three arguments and returns no value. Second and third arguments are optional. The function is used to draw a rectangle on the screen. The first argument is a *tuple* of four elements, where the first two elements are the *X* and *Y* position of the top left corner point of a rectangle, and the second two elements are the *X* and *Y* position of the bottom right corner point of a rectangle. The second argument of the `rectangle()` function is called “*outline*” and it represents the outline colour for edges of the rectangle. Third argument is called “*fill*” and it represents the colour for insides of the rectangle. If the `rectangle()` function is used without second and third arguments, the rectangle will be filled with black colour and the edges will be white.

Instead of the *tuple* for the first argument you can use `bounding_box` property of a *device* object, which returns *tuple* that points to the screen edges, or a rectangle that is as wide and as high as the screen dimensions.

Az-Delivery

You can use any of a library predefined colours (a predefined *Pillow* colour strings), or an RGB *tuple* of three elements, where every element is an integer number in range from 0 to 255, for example (255, 0, 0) (or in percentages (100%, 0%, 0%)) which is red colour. First element is the level of the colour red, the second argument is the level of the colour green and the third argument is the level of the colour blue. You can read more about colour options in the library documentation on the following link:

<https://pillow.readthedocs.io/en/3.1.x/reference/Imagecolour.html>

The *ellipse()* function accepts three arguments and returns no value. The function is used to draw an ellipse on the screen. First argument is a *tuple* of four elements, where the first two elements are the *X* and *Y* position of the top left corner point of a rectangle that surrounds the ellipse, and the second two elements are the *X* and *Y* position of bottom right corner point of a rectangle that surrounds the ellipse. The second argument of the *ellipse()* function is the *outline* argument, and the fourth argument is the *fill* argument. The *outline* and *fill* arguments are the same as the *outline* and *fill* arguments of the *rectangle()* function. These arguments are optional and the same rule for using *rectangle()* function without the second and third arguments applies to the *ellipse()* function.

Az-Delivery

The `polygon()` function accepts three arguments and returns no value. The function is used to draw a polygon (triangle or object with 3 or more corners) on the screen. First argument is a *list* that contains three or more *tuples*, where each *tuple* has two elements. Elements of specific *tuple* are the *X* and *Y* position of a specific point of the polygon. The polygon is created by connecting the *tuple* points, the first *tuple* point connects to the second, the second to the third, etc. Last *tuple* point in the *list* is connected to the first *tuple* point, therefore creating a polygon. The second argument of the `polygon()` function is the *outline* argument and the third argument is the *fill* argument. The *outline* and *fill* arguments are the same as the *outline* and *fill* arguments of the `rectangle()` function. These arguments are optional and the same rule for using `rectangle()` function without the second and third arguments, applies to the `polygon()` function.

The `line()` function accepts two arguments and returns no value. The second argument is optional. The function is used to draw a line on the screen. First argument is a *tuple* of four elements. First two elements are the *X* and *Y* position of the starting point of the line and the second two elements are the *X* and *Y* position of the ending point of the line. The second argument of the `line()` function is the *fill* argument. The *fill* argument is the same as the *fill* argument of the `rectangle()` function. If you use the `line()` function without this argument the colour of the line will be default, white.

Az-Delivery

The `text()` function accepts four arguments and returns no value. Third and fourth arguments are optional. The function is used to print text on the screen. First argument is a *tuple* of two elements where the elements are the *X* and *Y* position of the cursor. Second argument is a string, representing the text which will be printed on the screen. The third argument is the *font* argument which value is fixed-width font definition from the `luma.core.legacy.font` (we will explain fonts later in the text). The fourth argument is the *fill* argument (same as the *fill* argument of the `rectangle()` function). If you use `text()` function without third and fourth arguments, text will be white and the font will be set to *None*, indicating default font.

There are several predefined fonts in the `luma.core` library. The fonts can be found in the `luma.examples/examples/fonts` directory. Be careful with the fonts because many of them come with a specific licence. You can even create your own font, but we will not cover it in this eBook (you can read about it in the `luma.core` library documentation). To use the font from a `luma.core` library you have to follow a few steps. First you have to specify location to the font file (a file with extension “`.ttf`”), then you have to specify text size and then to use a function from the `PIL` library called `ImageFont.truetype()` to create the font. We do this in the following lines of the code:

```
font_path = '/home/pi/luma...../ChiKareGo.ttf'
size = 22
new_font = ImageFont.truetype(font_path, size)
```


AZ-Delivery

After this, we use *new_font* variable for font, like in the following line of the code:

```
draw.text((10, 70), 'AZ-Delivery', font=new_font, fill="red")
```

The self-defined function *changing_var()* accepts one argument and returns no value. The function is used to print a variable that changes its value. The argument value is the *device* object. At the beginning of the *changing_var()* function we create new font with size 40. Then in the *for* loop we change the value of a variable called "*i*". After that, we use the *device* object to create a *draw* object so that we can send commands to the screen:

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

Later, we print '*Changing variable.*' message on the screen with default font and check if the variable *i* is less than 10. If it is, we print leading zero before printing value of the variable. If the value of *i* variable is greater than 10 we just print the value of a variable. At the end of the *for* loop we inserted waiting period of one millisecond (`time.sleep(0.02)`). If you do not do this, the changes on the screen will be too fast for you to see them.

Az-Delivery

Then, we create *try-except* block where we wait for keyboard interrupt (*except KeyboardInterrupt*). When you press “CTRL + C” on the keyboard, keyboard interrupt happens and *except* block of the code will be executed, cleaning-up the object device and printing ‘*Script end!*’ message in the terminal.

In a *try* block of the code we create the infinite loop (*while True:*).

The infinite loop block is used to run all functions that we created. At the beginning of the infinite loop block we print the message “*Testing printing variable.*”, execute the *changing_var()* function and wait for two seconds. Then, we print the message “*Testing basic graphic*”, create the *draw* object, execute *primitives()* function and wait for three seconds. After that, we print the message “*Testing display ON/OFF*” and create the *for* loop (with five loops). In the *for* loop, we turn *OFF* the screen, wait for a half second, then turn *ON* the screen and wait for a half second. At the end of the infinite loop block we print the message “*Testing clear display.*”, wait for two seconds and execute *clear()* function. This function clears the data buffer of the screen.



Fonts from the Adafruit_GFX library

... > Arduino > libraries > Adafruit_GFX_Library > Fonts

FreeMono12pt7b.h	FreeSansBoldOblique12pt7b.h
FreeMono18pt7b.h	FreeSansBoldOblique18pt7b.h
FreeMono24pt7b.h	FreeSansBoldOblique24pt7b.h
FreeMono9pt7b.h	FreeSansBoldOblique9pt7b.h
FreeMonoBold12pt7b.h	FreeSansOblique12pt7b.h
FreeMonoBold18pt7b.h	FreeSansOblique18pt7b.h
FreeMonoBold24pt7b.h	FreeSansOblique24pt7b.h
FreeMonoBold9pt7b.h	FreeSansOblique9pt7b.h
FreeMonoBoldOblique12pt7b.h	FreeSerif12pt7b.h
FreeMonoBoldOblique18pt7b.h	FreeSerif18pt7b.h
FreeMonoBoldOblique24pt7b.h	FreeSerif24pt7b.h
FreeMonoBoldOblique9pt7b.h	FreeSerif9pt7b.h
FreeMonoOblique12pt7b.h	FreeSerifBold12pt7b.h
FreeMonoOblique18pt7b.h	FreeSerifBold18pt7b.h
FreeMonoOblique24pt7b.h	FreeSerifBold24pt7b.h
FreeMonoOblique9pt7b.h	FreeSerifBold9pt7b.h
FreeSans12pt7b.h	FreeSerifBoldItalic12pt7b.h
FreeSans18pt7b.h	FreeSerifBoldItalic18pt7b.h
FreeSans24pt7b.h	FreeSerifBoldItalic24pt7b.h
FreeSans9pt7b.h	FreeSerifBoldItalic9pt7b.h
FreeSansBold12pt7b.h	FreeSerifItalic12pt7b.h
FreeSansBold18pt7b.h	FreeSerifItalic18pt7b.h
FreeSansBold24pt7b.h	FreeSerifItalic24pt7b.h
FreeSansBold9pt7b.h	FreeSerifItalic9pt7b.h



You've done it!

Now you can use your module for various projects.

Now is the time to learn and make the Projects on your own. You can do that with the help of many example scripts and other tutorials, which you can find on the internet.

If you are looking for the high quality microelectronics and accessories, 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>