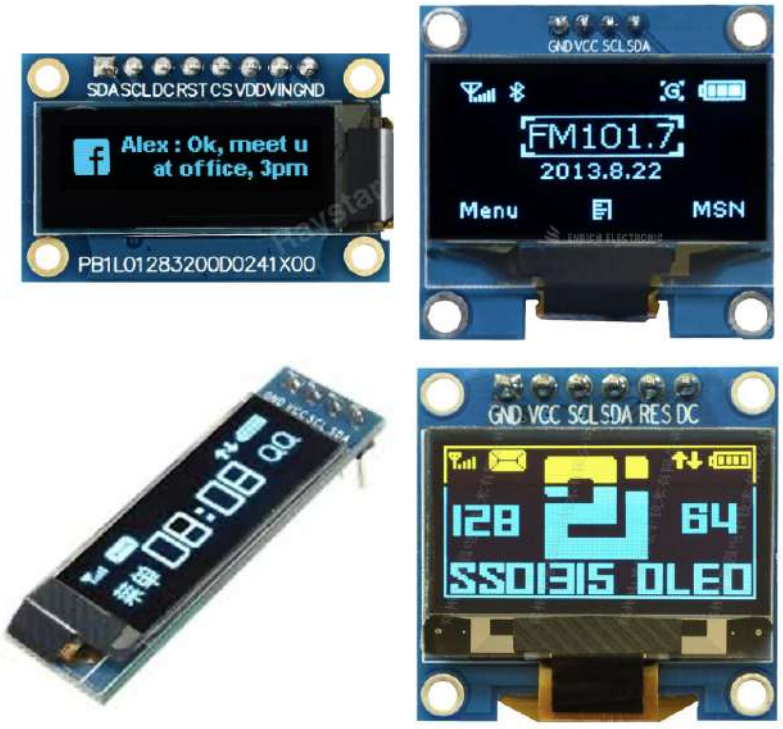
Одинадцатое практическое занятие

Работа с OLED экраном на чипе ssd1306

SSD1306 — это популярный контроллер для OLED-дисплеев с разрешением 128x64 или 128x32 пикселей.

Эти дисплеи широко используются в проектах благодаря своей компактности, низкому энергопотреблению и высокой контрастности.

Дисплеи на базе SSD1306 поддерживают интерфейсы I2C и SPI, но чаще всего используется I2C из-за простоты подключения.



Библиотека для работы с экраном на ssd1306

Работа с экраном на чипе ssd1306 требует загрузки на микроконтроллер дополнительного модуля.

Не будем переизобретать велосипед и воспользуемся существующей реализацией классов, скачать которые можно по следующим ссылкам:

- Ссылка на пакет для работы с ssd1306 (Яндекс Диск)

- Ссылка на пакет для работы с ssd1306 (GitHub)

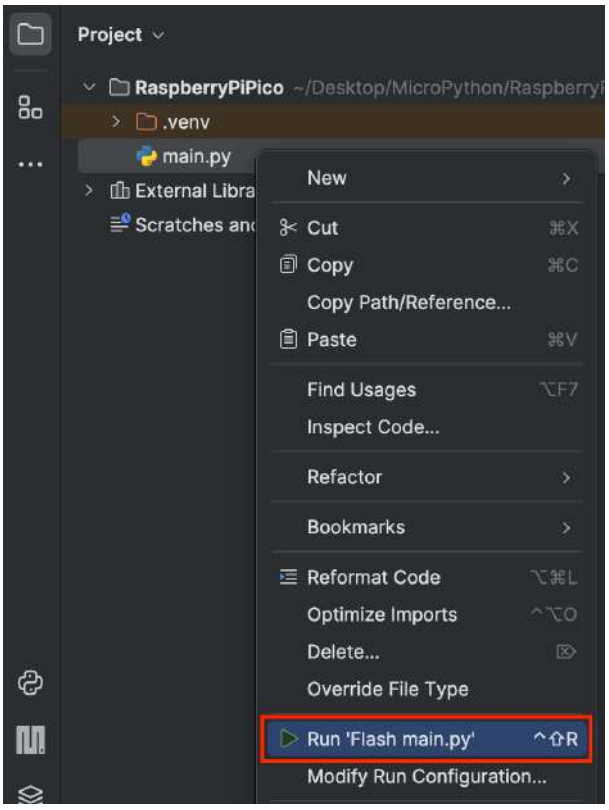
В результате у Вас на компьютере должен оказаться файл `ssd1306.py`!

Но что же делать с этим файлом и как заставить его работать с пользой для нас? Нужно загрузить его на микроконтроллер!

Но как это сделать так, чтобы иметь к нему удобный доступ? Для ответа на этот вопрос чуть отвлечемся и разберемся со стандартным модулем `os`, входящим в состав пакета `micropython`!

Модуль `os`

Если мы в чем-то уверены, так это в том, что выполнение любой программы на микроконтроллере RP Pico начинает выполняться с файла `main.py`, который необходимо загрузить в память из IDE (в нашем случае PyCharm):



Но, что произойдет, если мы подобным образом поступим с файлами с другими именами или находящимися в других директориях (не в корне проекта)?

Они также будут записаны в память микроконтроллера, но не будут исполняться без явного вызова их в коде файла `main.py`!

Наверняка внутри наших микроконтроллеров скопилось уже немало такого "балласта"... Но как узнать это? Нужно обратиться к памяти платы и прочитать ее!

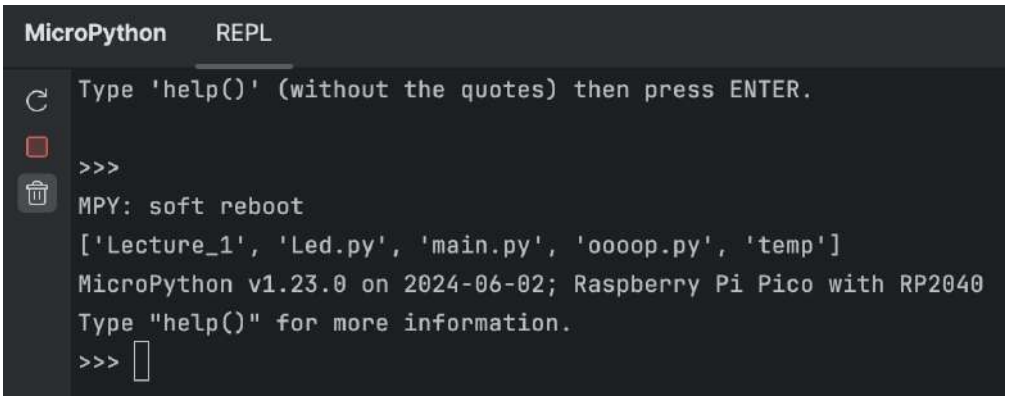
Это можно сделать, используя стандартный модуль `os`, документацию к которому можно найти по ссылке.

> Модуль `os` используется для манипуляций с *операционной системой*. Используемый модуль из `micropython` является упрощенной версией модуля `os` `python` (документация).

Для вывода в консоль списка файлов и директорий, находящихся в корне файловой системы микроконтроллера, выполним следующий код:

```
import os

print(os.listdir())
```

Выполнив его, мы сможем увидеть в консоли что-то наподобие этого:

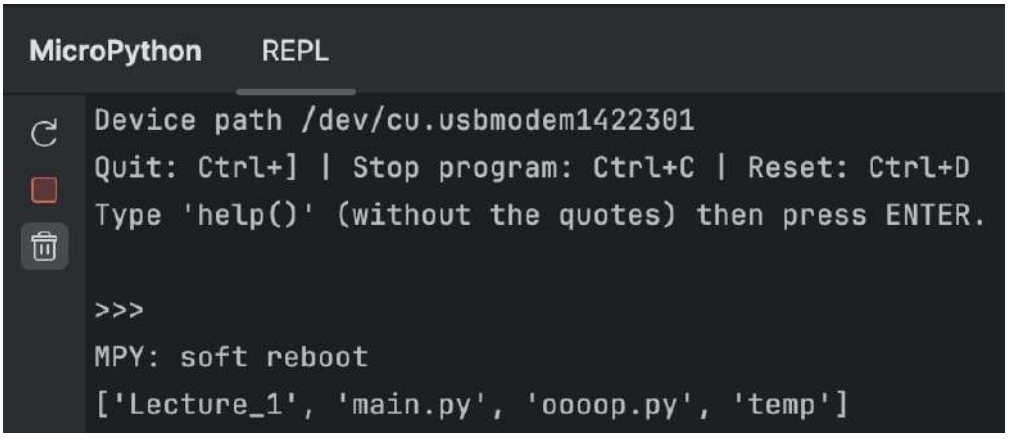


Список `['Lecture_1', 'Led.py', 'main.py', 'oooop.py', 'temp']` содержит в себе перечень всех файлов (те имена, которые имеют расширения) и папок записанных в память.

Удалить файл можно, использовав функцию `remove()` из модуля `os`, передав в качестве аргумента имя удаляемого файла.

Попробуем удалить таким образом файл `Led.py`

```
import os

os.remove('Led.py')
print(os.listdir())
```

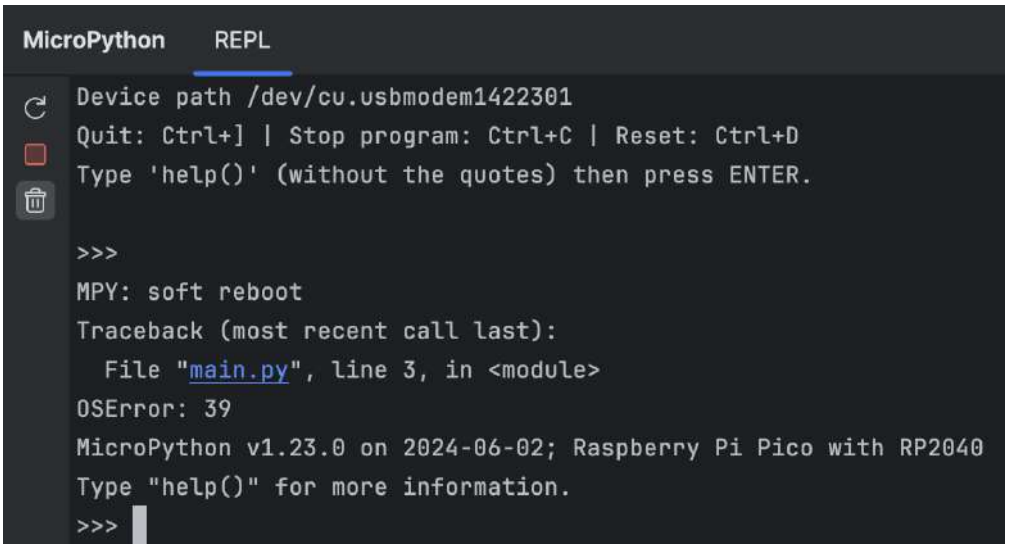


Вроде все прошло удачно - файл `Led.py` удален из памяти (отметим, что перезапуск кода будет теперь вызывать ошибку, вызванную тем, что файла с таким именем уже нет в памяти).

Но, что произойдет, если мы попробуем удалить подобным образом папку (например `Lecture_1`)?

```
import os

os.remove('Lecture_1')
print(os.listdir())
```

Мы получим только какую-то странную ошибку `OSError: 39` , но сама папка удалена не будет:



> Это связано с тем, что нельзя удалить папки напрямую, если в них содержатся файлы!

Не углубляясь излишне в утомительные подробности, просто приведем следующий код, позволяющий рекурсивно удалить все файлы, записанные в память микроконтроллера:

```
import os

def recursive_delete(path):
    # Получаем список всех элементов в текущей директории
    for item in os.listdir(path):
        # Формируем полный путь к элементу
        full_path = path + '/' + item
        # Если это файл, удаляем его
        if os.stat(full_path)[0] == 0x8000:  # 0x8000 означает, что это
файл
            os.remove(full_path)
            print(f"Удален файл: {full_path}")
        # Если это директория, рекурсивно удаляем её содержимое
        elif os.stat(full_path)[0] == 0x4000:  # 0x4000 означает, что это
директория
            recursive_delete(full_path)  # Рекурсивный вызов для удаления
содержимого
            os.rmdir(full_path)  # Удаляем пустую директорию
            print(f"Удалена директория: {full_path}")

# Пример использования: удаляем всё из корневой директории
recursive_delete('/')
```

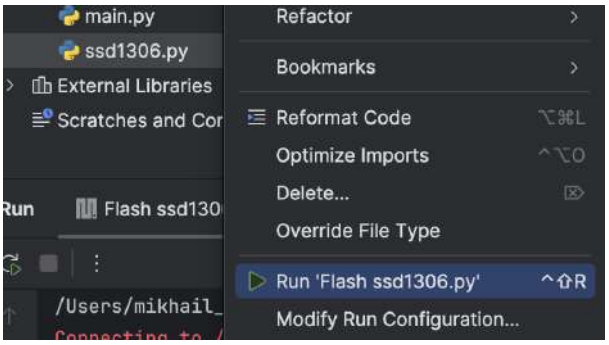
Но вернемся же к тому, ради чего все это начиналось, — загрузке ранее скаченного файла `ssd1306.py` в память микроконтроллера!

> ⓘ Строго говоря, для работы с экраном удалять все ранее записанные файлы было не обязательно, но так в нашей работе будет больше порядка, а, следовательно, и понятности!

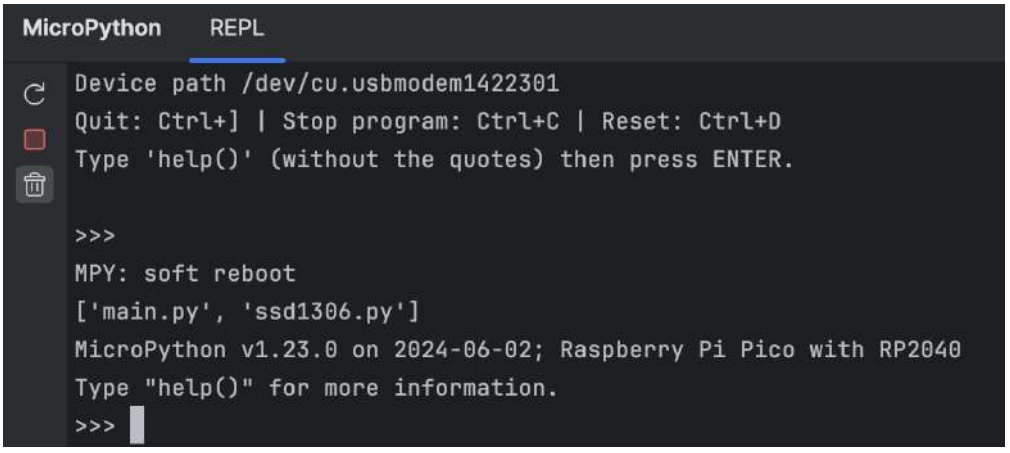
Установка пакета `ssd1306`

Запись модуля в память микроконтроллера

1. Файл `ssd1306.py` должен быть в корне Вашего проекта (на том же уровне, что и файл `main.py`).
2. Загрузите файл `ssd1306.py`:



3. Проверьте файлы через `os.lidtdir()



4. В результате в памяти должно быть два файла: `['main.py', 'ssd1306.py']`

Подключение модуля к файлу `main.py`

Файл `ssd1306.py` содержит в себе код трех классов:

- `SSD1306` - унаследованный от стандартного класса `micropython framebuf.FrameBuffer`;
- `SSD1306_I2C` и `SSD1306_SPI` - унаследованные от `SSD1306` и определяющие логику подключения экрана через **I2C** и **SPI** протоколы соответственно.

Загрузка необходимых для работы классов осуществляется через стандартные команды импорта в начале `main.py` файла:

```
from ssd1306 import SSD1306_I2C
```

Таким образом мы загрузили класс `SSD1306_I2C` для работы с экраном, подключенным по **I2C соединению**.

Самое время подключить экран!

Подключение экрана к RP Pico

Экран на `ssd1306` может подключаться через `I2C` и `SPI` протокол.

Мы в своей работе будем использовать I2C соединение!

I2C соединение

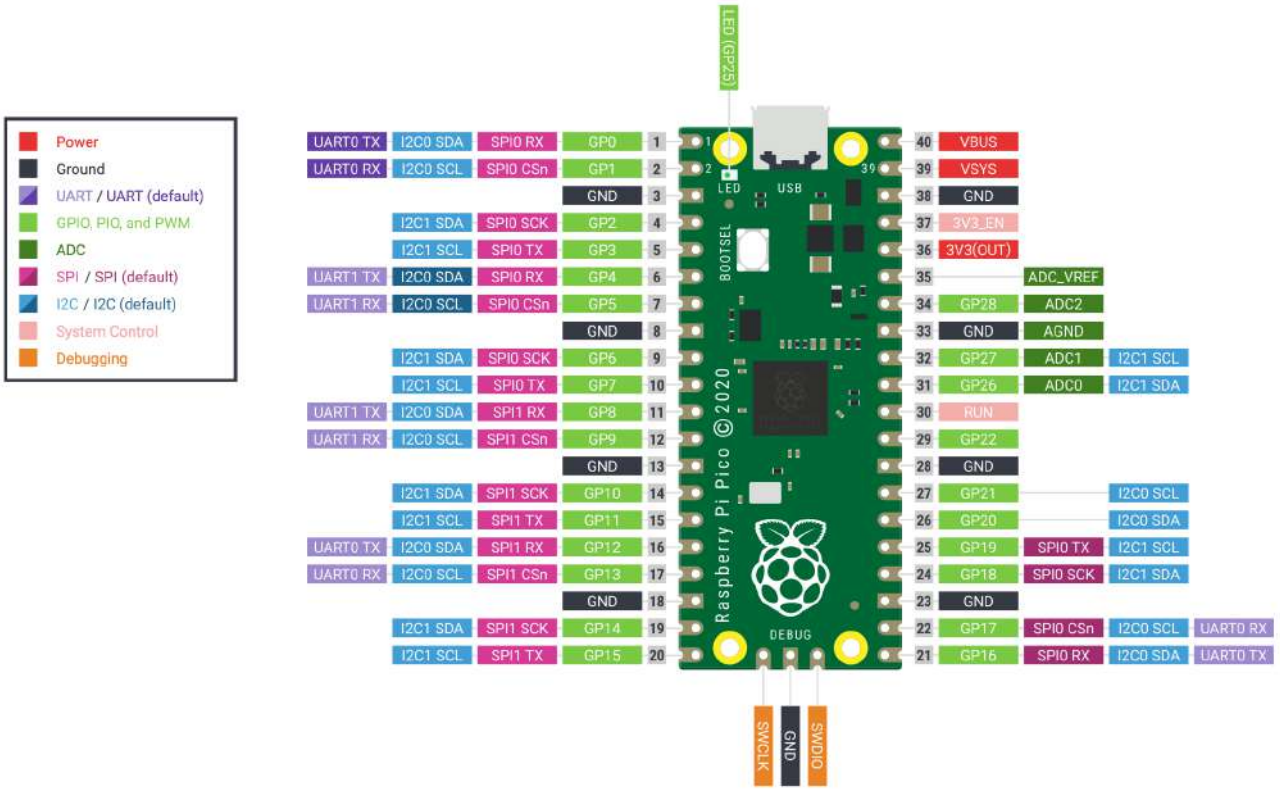
I²C (Inter-Integrated Circuit) — это популярный протокол для связи между микроконтроллерами и периферийными устройствами, такими как датчики, дисплеи, EEPROM и другие. В MicroPython реализована поддержка I²C, что позволяет легко работать с этим протоколом на таких платах, как ESP32, ESP8266, Raspberry Pi Pico и других.

I²C использует две линии для связи:

- **SCL (Serial Clock)** — линия тактового сигнала.
- **SDA (Serial Data)** — линия для передачи данных.

Устройства на шине I²C имеют уникальные адреса (обычно 7-битные), которые позволяют микроконтроллеру обращаться к конкретному устройству.

В RP Pico есть две I2C шины `I2C0` `I2C1` (выделены на схеме голубым цветом):



Для работы с I²C в MicroPython используется класс `machine.I2C`.

Вот пример настройки и использования I²C:

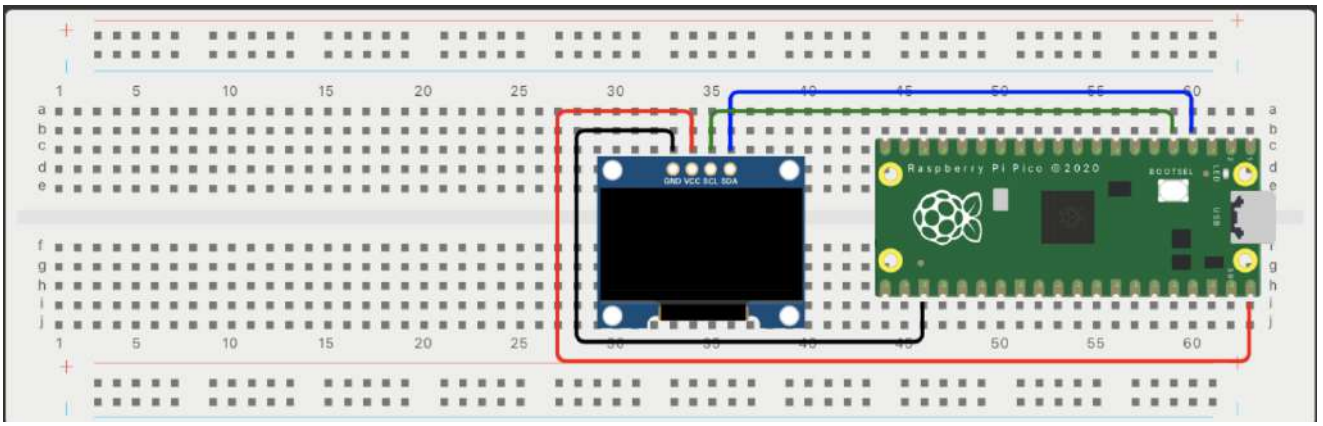
```
from machine import Pin, I2C

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )
```

Преимуществом I2C соединения является то, что к одной шине могут быть одновременно подключены несколько различных устройств, поддерживающих этот протокол связи.

Определение конкретного устройства выполняется в соответствии с его уникальным IP адресом.

Схема подключения

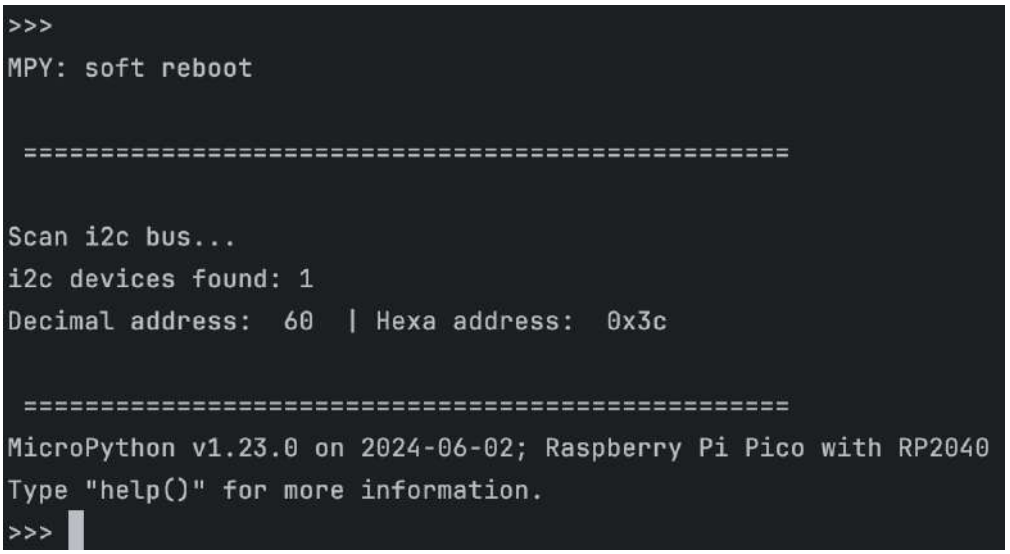
Подключим экран к первой I2C шине через пины 2 и 3 (можно к любым другим в соответствии со схемой GPIO)



проверим подключение и работоспособность экрана, просканировав подключенные к I2C шине устройства:

```
import machine

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = machine.I2C(id=I2C_ID,
                  sda=machine.Pin(SDA_PIN),
                  scl=machine.Pin(SCL_PIN))
print("\n", "=" * 50, "\n")
print('Scan i2c bus...')
devices = i2c.scan()

if len(devices) == 0:
    print("No i2c device !")
else:
    print('i2c devices found:', len(devices))

    for device in devices:
        print("Decimal address: ", device, " | Hexa address: ",
hex(device))
print("\n", "=" * 50)
```

В результате можем увидеть в консоли:



Круто! Одно устройства по 60 адресу (ox3c в шестнадцатеричной системе исчисления) было обнаружено, а значит с ним можно начинать работу!

Вывод информации на экран

Преамбула файла `main.py`

Создадим объект `SSD1306_I2C` для чего:

- импортируем соответствующий класс из файла `ssd1306`, записанного в память микроконтроллера;
- создадим объект класса I2C, отвечающий за соединение по конкретным пинам `SDA` и `SCL`
- создадим непосредственно объект для работы с экраном, передав в него количество пикселей по горизонтали (128) и вертикали(64), а также объект соединения (i2c).

Все это будет выглядеть примерно так:

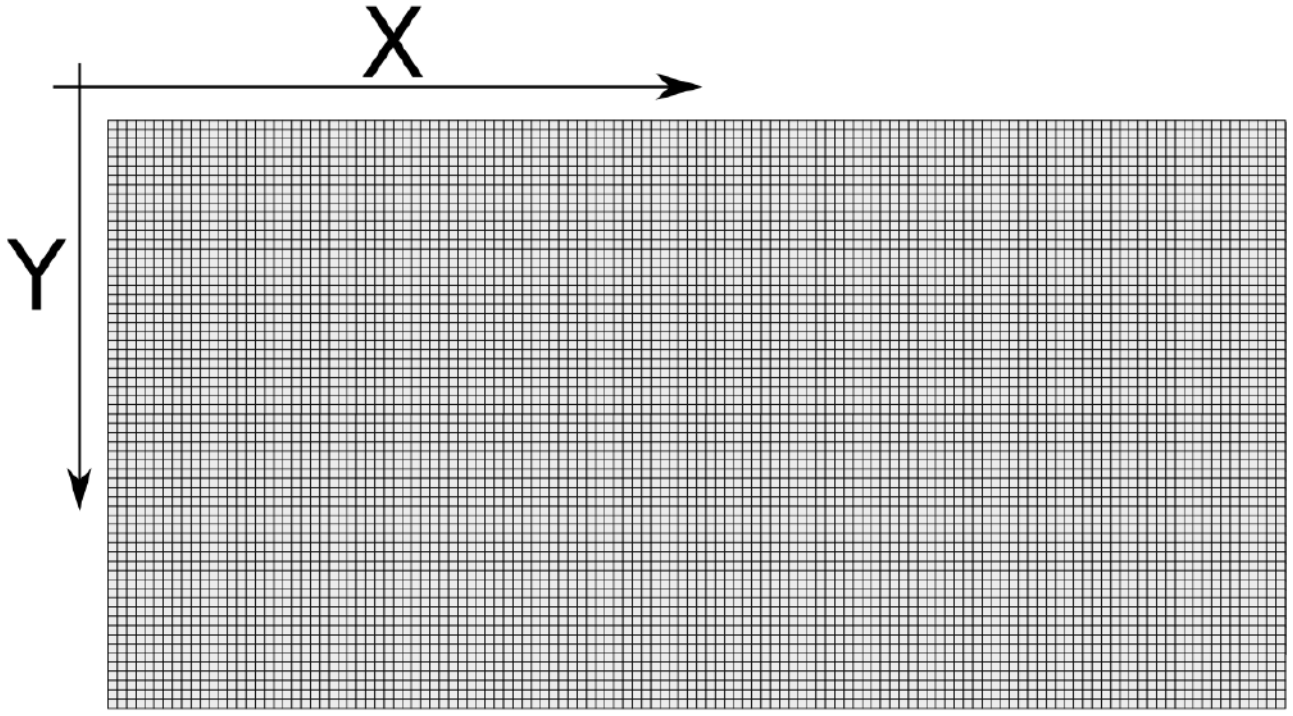
```
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
import time

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

display = SSD1306_I2C(128, 64, i2c)
```

```
time.sleep(1)
```

Задержка в 1 секунду в конце нужна для того, чтобы микроконтроллер успел наладить соединение с экраном и инициализировать все объекты.

Система координат экрана

Разрешение экрана 64x128 пикселей. Начало координат обычно делают в верхнем левом углу. X - вправо Y - вниз.



Основные методы работы с экраном

Обновление экрана

После выполнения любых изменений на экране нужно вызвать метод `show()`, чтобы применить их.

```
display.show()
```

Очистка экрана

Метод `fill()` заполняет весь экран указанным цветом (0 — черный, 1 — белый). После изменения экрана нужно вызвать `show()`, чтобы обновить дисплей.

```
display.fill(0)  # Очистка экрана (черный цвет)
display.show()   # Применение изменений
```

```
display.fill(1)  # Очистка экрана (белый цвет)
display.show()   # Применение изменений
```

Отображение текста

Метод `text()` позволяет выводить текст на экран. Нужно указать текст, координаты (x, y) и цвет.

```
display.fill(0)
display.text('Hello!', 0, 0, 1)  # Текст, x, y, цвет (1 — белый)
display.show()
```

По умолчанию класс `SSD1306_I2C` может работать только с латинскими буквами - **кириллические символы отрисовываться не будут!**

> Если Вы чувствуете себя уверенными в теме наследования и полиморфизма и у Вас есть свободный досуг, Вы сможете исправить этот досадный недостаток!

Одновременно можно добавить на экран несколько строк текста:

```
display.fill(0)
display.text('Hello, World!', 0, 0, 1)
display.text('SSD1306', 0, 16, 1)
display.text('Mining University', 32, 32, 1)
display.show()
```

> При этом, если строка длинная, текст может выйти за пределы экрана!

В таком случае лучше разбить длинную строку на несколько коротких:

```
display.fill(0)
display.text('Hello, World!', 0, 0, 1)
display.text('SSD1306', 0, 16, 1)
display.text('Mining', 32, 32, 1)
display.text('University', 32, 48, 1)
display.show()
```

Можно писать "черным" по "белому":

```
display.fill(1)
display.text('Hello, World!', 0, 0, 0)
display.text('SSD1306', 0, 16, 0)
display.text('Mining', 32, 32, 0)
```

```
display.text('University', 32, 48, 0)
display.show()
```

Если ошибиться с координатами и обновлением экрана, то текст на экране будет накладываться:

```
display.fill(0)

display.text('Hello, World!', 0, 0, 1)
display.text('SSD1306', 0, 16, 1)

display.text('Mining', 32, 32, 1)
display.show()
time.sleep(1)

display.text('University', 32, 32, 1)
display.show()
```

Исправить это можно через "очистку" экрана:

```
display.fill(0)
display.text('Hello, World!', 0, 0, 1)
display.text('SSD1306', 0, 16, 1)
display.text('Mining', 32, 32, 1)
display.show()

time.sleep(1)

display.fill(0)
display.text('Hello, World!', 0, 0, 1)
display.text('SSD1306', 0, 16, 1)
display.text('University', 32, 32, 1)
display.show()
```

Рисование пикселя

Метод `pixel()` рисует точку на экране. Нужно указать координаты (x, y) и цвет.

```
display.fill(0)
display.pixel(10, 10, 1)  # Рисует белую точку на координатах (10, 10)
display.show()
```

Рисование линии

Метод `line()` рисует линию между двумя точками. Нужно указать координаты начала (x1, y1), конца (x2, y2) и цвет.

Произвольная линия

```
display.line(0, 0, 127, 63, 1)  # Линия от (0, 0) до (127, 63), белый цвет
display.show()
```

Горизонтальная линия

```
display.fill(0)

display.hline(20, 20, 30, 1) # Горизонтальная линия от (20, 20), длинной в 30 пикселей, белый цвет
display.hline(20, 40, 2, 1)  # Горизонтальная линия от (20, 40), длинной в 2 пикселя, белый цвет
display.hline(20, 60, 1, 1)  # Горизонтальная линия от (20, 40), длинной в 1 пиксель, белый цвет

display.show()
```

Вертикальная линия

```
display.fill(0)

display.vline(20, 20, 30, 1) # Верительная линия от (20, 20), длинной в 30 пикселей, белый цвет
display.vline(40, 20, 2, 1)  # Верительная линия от (40, 20), длинной в 2 пикселя, белый цвет
display.vline(60, 20, 1, 1)  # Верительная линия от (60, 20), длинной в 1 пиксель, белый цвет

display.show()
```

Рисование полилинии

Полилиния отрисовывается с помощью метода `poly()`, который принимает координаты вставки (в примере (0, 0)) массива `array`.
Следующие параметры определяют цвет полилинии и необходимость ее заливки:

```
import array

parallelogram = array.array('I', [20, 20, 30, 30, 30, 60, 20, 50])
display.poly(0, 0, parallelogram, 1, 1)
display.show()
time.sleep(1)

shadow = array.array('I', [20, 20, 30, 30, 30, 60, 20, 50])
display.poly(60, 0, shadow, 1, 0)
```

```
display.show()
time.sleep(1)
```

сам массив: `array.array('I', [20, 20, 30, 30, 30, 60, 20, 50])` определяет последовательность координат точек $(20, 20) - (30, 30) - (30, 60) - (20, 50)$

Рисование прямоугольника

Метод `rect()` рисует прямоугольник. Нужно указать координаты верхнего левого угла (x, y), ширину, высоту и цвет.

Обычный прямоугольник

```
display.rect(10, 10, 50, 30, 1)  # Прямоугольник 50x30 на координатах (10, 10), белый цвет
display.show()
```

Заливка прямоугольника

Можно явно поставить флаг "заливки" в методе `rect()`.

```
display.rect(10, 10, 50, 30, 1, 1)  # Залитый прямоугольник 50x30, белый цвет
display.show()
```

Или использовать отдельный метод `fill_rect()` который рисует закрашенный прямоугольник. Параметры аналогичны `rect()`:

```
display.fill_rect(10, 10, 50, 30, 1)  # Залитый прямоугольник 50x30, белый цвет
display.show()
```

Рисование окружности

По умолчанию метода для рисования окружности в модуле нет, но есть метод для рисования эллипса - `ellipse()`.

Очевидно, что, если задать в эллипсе равные полуоси, получим окружность:

```
display.fill(0)

display.ellipse(64, 42, 20, 20, 1) # Окружность с центром в (64, 42), радиусом в 20, белый цвет
display.show()
time.sleep(1)
```

```
display.ellipse(64, 42, 60, 20, 1) # Эллипс с центром в (64, 42), радиусом
по x=60, y=20, белый цвет
display.show()
time.sleep(1)

display.ellipse(64, 42, 5, 20, 1, 1) # Pfkbnsq 'ллипс с центром в (64,
42), радиусом по x=5, y=20, белый цвет
display.show()
time.sleep(1)

display.show()
```

Собственная функция для рисования окружности

Напишем свою функцию `circle()`, обладающую дополнительными возможностями по отрисовке окружности:

```
def circle(display, x_center, y_center, radius, thickness=1, color=1,
fill=False):
    if fill:
        thickness = radius
    for dr in range(thickness):
        r = radius - dr
        for degree in range(360):
            radians = math.radians(degree)
            x = r * math.cos(radians)
            y = r * math.sin(radians)
            display.pixel(int(x+x_center), int(y+y_center), color)
```

Так наша функция может отрисовывать окружность заданной толщины `thickness`

```


display = SSD1306_I2C(128, 64, i2c)
time.sleep(1)

display.fill(0)
circle(display=display,
       x_center=64,
       y_center=32,
       radius=30,
       thickness=5,
       color=1)

circle(display=display,
       x_center=64,
       y_center=32,
       radius=10,
       fill=True,
```

```
        color=1)

display.show()
```

Отображение изображения

Для того чтобы вывести на экран изображение, необходимо:

1. Записать изображение в массив байт (можно воспользоваться онлайн конвертерами, например этим)
2. Создать из этого массива объект `FrameBuffer`, указав размер изображения
3. Загрузить изображение в дисплей через метод `blit()`
4. Обновить отображение на экране

```
image = bytearray([0b00011000,
                   0b00111100,
                   0b01111110,
                   0b11011011,
                   0b11111111,
                   0b00100100,
                   0b01011010,
                   0b10100101])
import framebuf
# Load the image into a framebuffer (the image is 8x8)
fb = framebuf.FrameBuffer(image, 8, 8, framebuf.MONO_HLSB)

# Draw the image at coordinates X=60 and Y=28
display.blit(fb, 60, 28)
display.show()
```

Пример изображения

Ссылка на файл с примером кода

Инверсия цвета

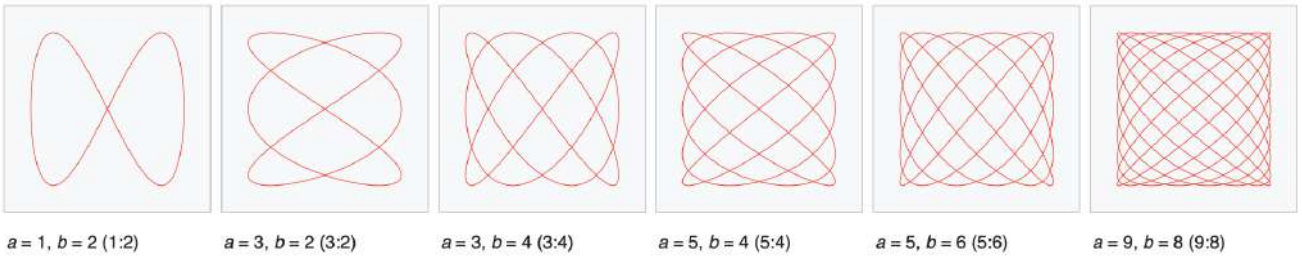
Метод `invert()` инвертирует цвета на экране (черный становится белым, и наоборот). Нужно передать 1 для включения инверсии и 0 для выключения.

```
display.invert(1)  # Включить инверсию
display.show()
```

Примеры

Фигуры Лиссажу

Фигуры Лиссажу - траектории, прочерчиваемые точкой, совершающей одновременно два гармонических колебания в двух взаимно перпендикулярных направлениях.




```
from math import sin, pi, radians

from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
import time

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

WIDTH = 128
HEIGHT = 64

display = SSD1306_I2C(WIDTH, HEIGHT, i2c)
time.sleep(1)

# Параметры фигуры Лиссажу
A = 25  # Амплитуда по X
B = 25  # Амплитуда по Y

f1 = 1  # Частота по X
f2 = 2  # Частота по Y

# Функция для отрисовки фигуры Лиссажу
def draw_lissajous(phase=0):
    display.fill(0)  # Очистка экрана
    for t in range(720):
        x = int(WIDTH / 2 + A * sin(f1 * t / 100))  # Координата X
        y = int(HEIGHT / 2 + B * sin(f2 * t / 100 + phase))  # Координата
Y
        display.pixel(x, y, 1)  # Рисуем точку
    display.show()  # Обновляем экран
```

```
# Основной цикл
while True:
    phase = 0
    for delta_phase in range(360):
        phase += radians(delta_phase)
        draw_lissajous(phase)  # Рисуем фигуру Лиссажу
        time.sleep(0.1)  # Пауза перед обновлением
```

Ссылка на код

Параметры фигуры Лиссажу:

```
- `A` и `B` – амплитуды колебаний по осям X и Y.
- `f1` и `f2` – частоты колебаний по осям X и Y.
- `phase` – фазовый сдвиг между колебаниями.
```

Окружность

```
f1 = 1
f2 = 1
```

Восьмёрка

```
f1 = 1
f2 = 2
```

Сложная фигура

```
f1 = 3
f2 = 2
```

Ультразвуковой дальномер с экраном

- Сохраните в файл `hcsr04.py` следующий код:

```
from machine import Pin
from time import ticks_us, sleep_us, ticks_diff


class HCSR04:

    __SPEED_OF_SOUND = 0.0343
    __IMPULSE_LAG = 10_000
```

```
    def __init__(self, trigger_pin, echo_pin):
        self.trigger = Pin(trigger_pin, Pin.OUT)
        self.echo = Pin(echo_pin, Pin.IN)
        self.signal_start = None
        self.signal_stop = None
        self.last_measure_time = ticks_us()
        self.echo.irq(trigger=Pin.IRQ_RISING | Pin.IRQ_FALLING,
handler=lambda pin: self._handler())

    def _handler(self):
        current_time = ticks_us()
        if self.echo.value() == 1:
            self.signal_start = current_time
        else:
            self.signal_stop = current_time

    def _send_signal(self):
        while ticks_diff(ticks_us(), self.last_measure_time) <
self.__IMPULSE_LAG:
            pass
        self.trigger.low()
        sleep_us(2)
        self.trigger.high()
        sleep_us(10)
        self.trigger.low()

    def _ultra_sonic_time_measure(self):
        self._send_signal()
        while self.echo.value() == 0:
            signal_start = ticks_us()
        while self.echo.value() == 1:
            signal_stop = ticks_us()
        time_passed = ticks_diff(signal_stop, signal_start)
        self.last_measure_time = ticks_us()
        return time_passed

    def get_distance(self, round_distance=True):
        pulse_duration = self._ultra_sonic_time_measure()
        distance_cm = (pulse_duration * self.__SPEED_OF_SOUND) / 2
        if round_distance:
            return round(distance_cm, 2)
        return distance_cm

    def get_median_distance(self, num_of_measures=3, round_distance=True):
        distances = [self.get_distance(round_distance=round_distance) for
_ in range(num_of_measures)]
        distances.sort()
        n = len(distances)
        if n % 2 == 1:
```

```
            return distances[n // 2]
        else:
            return (distances[n // 2 - 1] + distances[n // 2]) / 2
```

Ссылка на код

и запишите его в память микроконтроллера аналогично файлу `ssd1306.py`

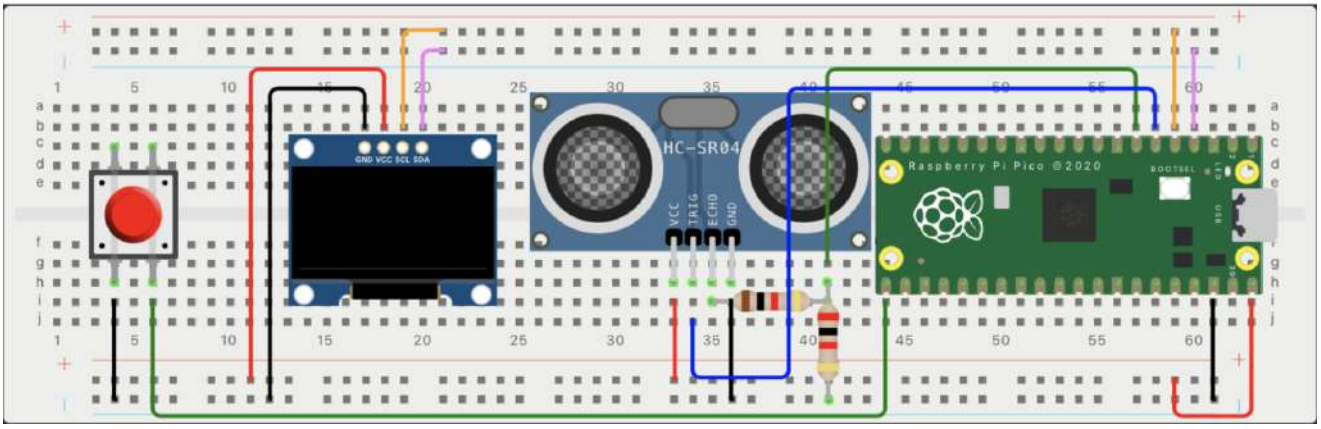
- Повторите все это для классов кнопок (файл `button.py`):

```
from machine import Pin
from time import ticks_ms


class Button:
    def __init__(self, button_pin_number, trigger=Pin.IRQ_FALLING,
pull=Pin.PULL_UP, debounce_time=50):
        self.pin = Pin(button_pin_number, Pin.IN, pull)
        self.debounce_time = debounce_time
        self.last_state = self.pin.value()
        self.last_click_time = ticks_ms()
        self.action = False
        self._current_action = None
        self.pin.irq(trigger=trigger, handler=lambda pin:
self.check_button_click())

    def check_button_click(self):
        current_time = ticks_ms()
        if current_time - self.last_click_time > self.debounce_time:
            self.action = True
            self._current_action = "single"
            self.last_click_time = ticks_ms()

    @property
    def current_action(self):
        action = self._current_action
        self._current_action = None
        self.action = False
        return action


class MultyClickButton:
    def __init__(self, button_pin_number,
                 trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING,
                 pull=Pin.PULL_UP,
                 debounce_time=50, total_command_time=500):
        self.pin = Pin(button_pin_number, Pin.IN, pull)
        self.debounce_time = debounce_time
        self.total_command_time = total_command_time
```

```
        self.last_state = self.pin.value()
        self.last_change_time = ticks_ms()
        self.click_count = 0
        self.last_click_time = ticks_ms()
        self.action = False
        self._current_action = None
        self.pin.irq(trigger=trigger, handler=lambda pin:
self.check_button_state())

    @property
    def current_action(self):
        self.check_clicks()
        if self.action:
            action = self._current_action
            self._current_action = None
            self.action = False
            return action

    def check_button_state(self):
        current_time = ticks_ms()
        current_state = self.pin.value()
        if current_state != self.last_state:
            if current_time - self.last_change_time > self.debounce_time:
                self.last_state = current_state
                self.last_change_time = current_time
                if not current_state:
                    self.click_count += 1
                    self.last_click_time = current_time

    def check_clicks(self):
        current_time = ticks_ms()
        if self.pin.value() and self.click_count > 0:
            if current_time - self.last_click_time >
self.total_command_time:
                if self.action is False:
                    self.action = True
                    if self.click_count == 1:
                        self.click_count = 0
                        self._current_action = "single"
                    elif self.click_count == 2:
                        self.click_count = 0
                        self._current_action = "double"
                    elif self.click_count == 3:
                        self.click_count = 0
                        self._current_action = "triple"
                    else:
                        result = f"multiple - {self.click_count}"
                        self.click_count = 0
                        self._current_action = result
```

[Ссылка на код]

Схема подключения



Управляющий код файла `main.py`

```
from time import sleep_us

from machine import I2C, Pin

from hcsr04 import HCSR04
from button import MultyClickButton
from ssd1306 import SSD1306_I2C

button = MultyClickButton(button_pin_number=16)
hcsr04 = HCSR04(trigger_pin=4, echo_pin=5)

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

dsp = SSD1306_I2C(128, 64, i2c)

def print_text_on_screen(text_str):
    dsp.fill(0)
    dsp.text(text_str, 0, 32)
    dsp.show()

sleep_us(1_000_000)
print_text_on_screen("Ready!")


while True:
```

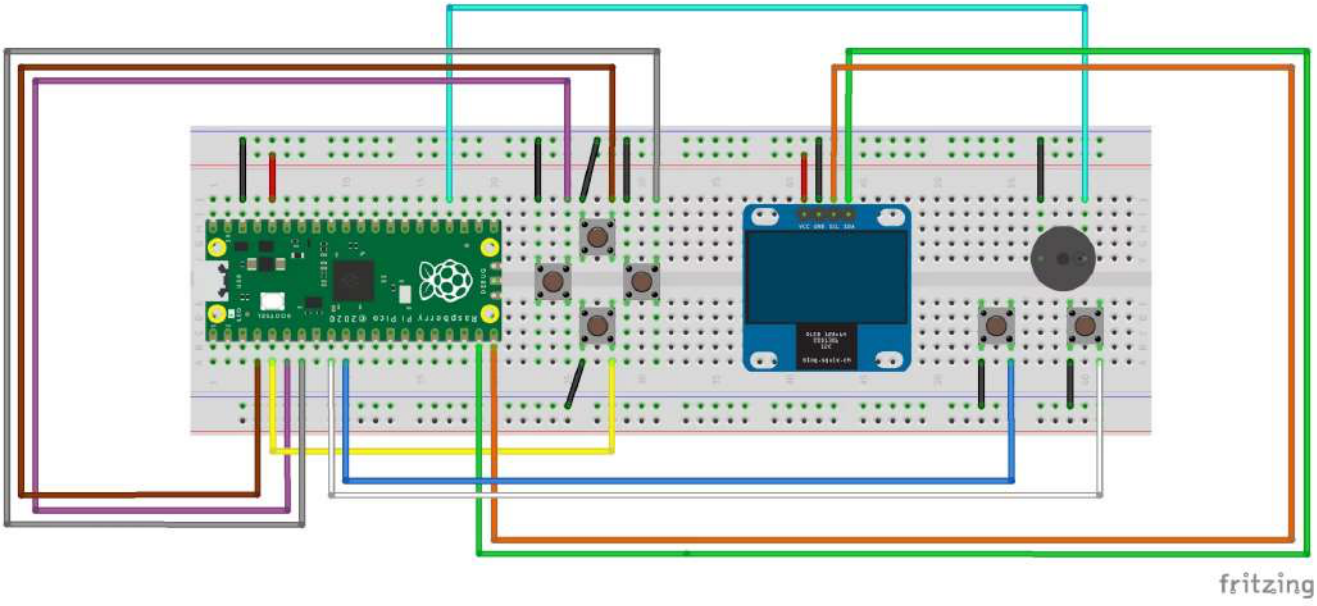
```
    bs = button.current_action
    if bs == "single":
        msg = f"Dictance: {hcsr04.get_distance()} cm"        print(msg)
        print_text_on_screen(msg)
    if bs == "double":
        msg = f"Median: {hcsr04.get_median_distance()} cm"
print(msg)
        print_text_on_screen(msg)
    if bs == "triple":
        distances = [hcsr04.get_median_distance(num_of_measures=3,
round_distance=False) for _ in range(10)]
        distance = round(sum(distances) / len(distances), 2)
        msg = f"Average {distance} cm"        print(msg)
        print_text_on_screen(msg)
    sleep_us(50_000)
```

Raspberry Pi Pico Retro Gaming System

Ссылка на страницу проекта

[Страница проекта]

Схема подключения



> В иходном проекте пины для подключения кнопок и экрана прописаны "хардкодом" - поэтому подключать все нужно именно по приведенной схеме или менять сам код!

Ссылки на исходный код проекта

Ссылки на GitHub

Ссылка на репозиторий в GitHub

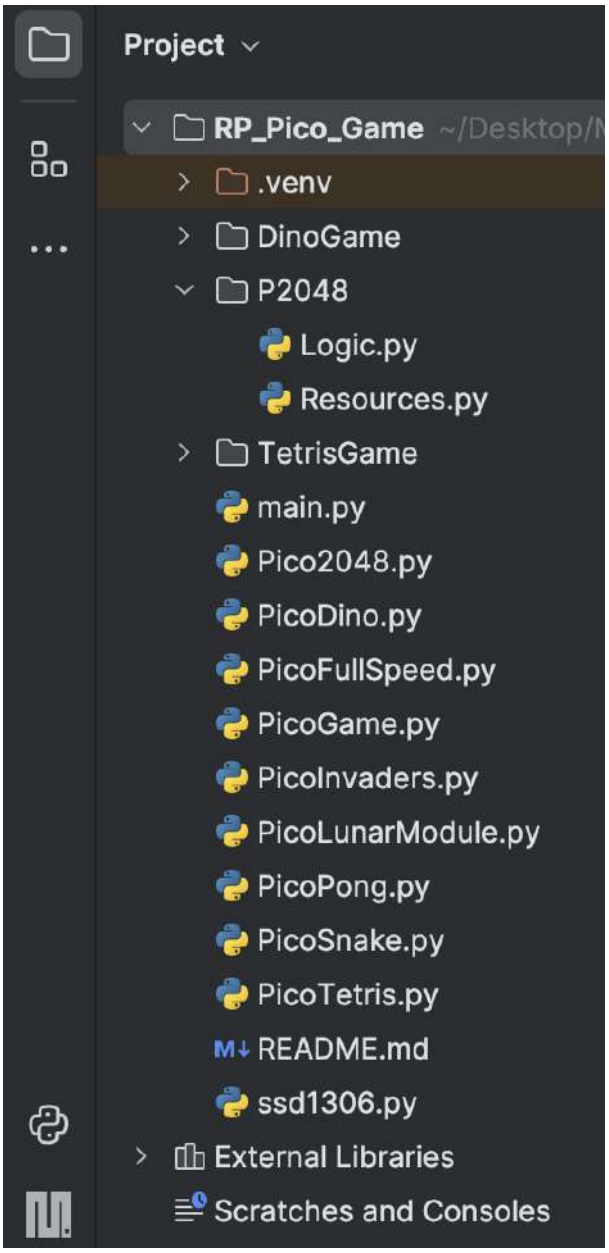
⚠ В исходном коде с GitHub есть несколько опечаток, исправьте их самостоятельно или возьмите код с YandexDisk!

Ссылки на YandexDisk

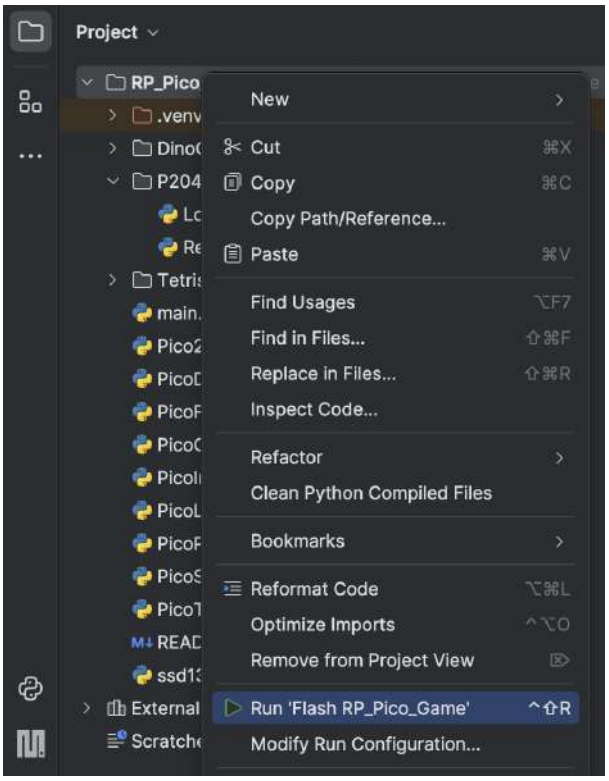
Ссылка на архив YandexDisk

Установка кода

Распакуйте и сохраните все файлы в корень проекта `PyCharm` не меняя их структуры:



Нажмите правой кнопкой мыши по самому проекту и выберете его загрузку на микроконтроллер:


Project
RP_Pico
.venv
Dino(
P204
Lc
Re
Tetris
main.
Pico2
PicoD
PicoF
PicoC
Picoli
PicoL
PicoF
PicoS
PicoT
READ
ssd13
External
Scratche
New
Cut ⌘X
Copy ⌘C
Copy Path/Reference...
Paste ⌘V
Find Usages ⌥F7
Find in Files... ⇧⌘F
Replace in Files... ⇧⌘R
Inspect Code...
Refactor
Clean Python Compiled Files
Bookmarks
Reformat Code ⌥⌘L
Optimize Imports ^⌥O
Remove from Project View
Run 'Flash RP_Pico_Game' ^⇧R
Modify Run Configuration...