Девятое практическое занятие

Прерывания и таймеры

Введение

Мы уже уделили достаточно много внимания разработке и работе с неблокирующими основной код программы классами - кнопками, диодами и т.п.

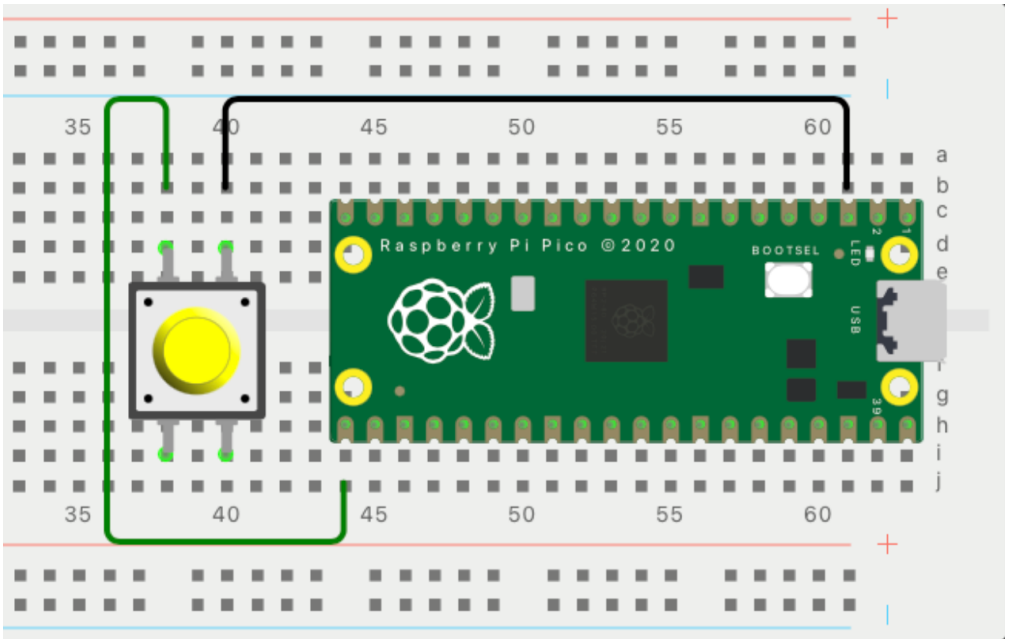
Во всех изученных случаях необходимость такого подхода диктовалась тем, что задержки процессов, выполняемых в определенный момент определённым элементом устройства, нарушает корректную работу остальных его элементов, что, очевидно, не допустимо.

В рассмотренных к текущему моменту примерах нас выручала замена прямых задержек выполнения кода (функций подобных `sleep()` из модуля `time`) проверкой прошедшего с предыдущего события интервала времени (наподобие:

```
if current_time - self.last_change_time > self.debounce_time:
            self.last_change_time = current_time
            ...
```

)

Хотя, часто такое решение является необходимым и неизбежным, абсолютной панацеей, решающей все проблемы, оно, к сожалению, не является.
Проиллюстрируем это на следующем примере с простой кнопкой:



```
from machine import Pin
from time import ticks_ms, sleep


class Button:
    def __init__(self, button_pin_number, debounce_time=50):
        self.pin = Pin(button_pin_number, Pin.IN, Pin.PULL_UP)
        self.debounce_time = debounce_time
        self.last_state = self.pin.value()
        self.last_change_time = ticks_ms()

    def check_button_click(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:
                    return "Click!"


BUTTON_PIN = 16
button = Button(button_pin_number=BUTTON_PIN)

counter = 0
while True:
    action = button.check_button_click()
    if action:
        print(counter, ticks_ms(), action)
        counter += 1
    sleep(0.01)
    # sleep(1)
    # sleep(10)
```

С представленным кодом мы уже работали на предыдущем занятии и никаких проблем не было...

В текущей версии их нет и сейчас, если оставить задержку `sleep(0.01)` в основном цикле программы.

Но что случится, если ее увеличить? До секунды или даже больше... Это легко проверить, раскомментировав соответствующие строки кода.

Сделав это, нам придется признать, что кнопка перестала нормально отрабатывать нажатия на нее.

Очевидно, это вызвано тем, что во время "сна" микроконтроллера опрос кнопки в строке `action = button.check_button_click()` не происходит. В реальном коде никто не станет (надеюсь) использовать такие длинные задержки, но, если последовательно с опросом кнопки будет выполняться какой-либо сложный, ресурсоемкий, а, следовательно, и долгий по времени исполнения, код, эффект будет тем же.

Говорят, первый шаг к решению проблемы осознать ее... Если это сделано, перейдем к следующему - как добиться возможности реакции микрокотроллера на нажатие кнопки независимо от текущего места выполнения кода.

Как всегда, добиться этого можно несколькими способами, распараллеливающими процесс выполнения кода, используя:

- многопоточность;
- прерывания;
- таймеры.

Среди перечисленного многопоточный код (разделяющий выполнения команд между физически разными ядрами процессора) мы отложим на потом, а пока изучим менее сложные способы!

Прерывание

Прерывание в микроконтроллере — это механизм, который позволяет микроконтроллеру мгновенно реагировать на внешние или внутренние события, приостанавливая выполнение текущей программы и переходя к выполнению специальной функции, называемой **обработчиком прерывания** (или ISR — Interrupt Service Routine). После завершения обработки прерывания микроконтроллер возвращается к выполнению основной программы.

Основные понятия:

1. **Источники прерываний**:
    - **Внешние прерывания**: Вызываются внешними событиями, например, нажатием кнопки, изменением уровня сигнала на входе микроконтроллера.
    - **Внутренние прерывания**: Генерируются внутренними периферийными устройствами микроконтроллера, например, таймерами, UART, ADC и т.д.
2. **Обработчик прерывания (ISR)**:
    - Это функция, которая выполняется при возникновении прерывания.
    - Она должна быть короткой и быстрой, чтобы не задерживать выполнение основной программы.
3. **Приоритет прерываний**:
    - Некоторые прерывания могут иметь более высокий приоритет, чем другие. Если два прерывания происходят одновременно, сначала обрабатывается

прерывание с более высоким приоритетом.

4. **Маскирование прерываний**:
    - Прерывания можно временно отключать (маскировать), чтобы предотвратить их обработку в критических участках кода.

Как работает прерывание:

1. **Возникновение события**:
    - Происходит событие, которое может вызвать прерывание (например, нажатие кнопки или переполнение таймера).
2. **Запрос на прерывание**:
    - Источник прерывания отправляет запрос на прерывание в процессор микроконтроллера.
3. **Приостановка основной программы**:
    - Микроконтроллер сохраняет текущее состояние (например, значения регистров) и приостанавливает выполнение основной программы.
4. **Выполнение обработчика прерывания**:
    - Микроконтроллер переходит к выполнению обработчика прерывания (ISR).
5. **Возврат к основной программе**:
    - После завершения обработки прерывания микроконтроллер восстанавливает сохраненное состояние и продолжает выполнение основной программы с того места, где она была прервана.

Рассмотрим базовый синтаксис прерывания:

```
from machine import Pin
import time

BUTTON_PIN = 16
button = Pin(BUTTON_PIN, Pin.IN, Pin.PULL_UP)


# Функция-обработчик прерывания
def button_pressed(pin):
    print("Кнопка нажата!")

# counter = 0

# def button_pressed(pin):
#     global counter
#     print(counter, "Кнопка нажата!")
#     counter += 1

# Настраиваем прерывание на срабатывание при падающем фронте (нажатие
кнопки)
```

```
button.irq(trigger=Pin.IRQ_FALLING, handler=button_pressed)


while True:
    time.sleep(10)
```

Прерывание непосредственно настраивается на конкретном пине (в нашем примере на 16 пине, на объект которого ссылается переменная `button`)

Настройка режима прерывания выполняется в методе пина `irq`, принимающим в себя два атрибута: `trigger` - определяющий момент срабатывания прерывания и `handler` - принимающий функцию, которая будет выполняться после наступления события `trigger`.

```
button.irq(trigger=Pin.IRQ_FALLING, handler=button_pressed)
```

В нашем примере `Pin.IRQ_FALLING` определяет момент прерывания, когда высокий сигнал на пине (логическая единица - напряжение 3.3V) упадет до низкого (логический ноль - 0V). В примере с нашей кнопкой это наступает в момент ее нажатия, после чего прерывание запустит функцию, переданную в переменную `handler` (в нашем примере `aeyrwb. button_pressed`)

> Отметим, что при инициализации прерывания передается именно объект функции - без скобок!

Рассмотрим подробнее функцию `button_pressed`:

```
# Функция-обработчик прерывания
def button_pressed(pin):
    print("Кнопка нажата!")
```

В первую очередь отметим, что в обработчик прерывания **обязательно передается сам объект пина** в качестве обязательного аргумента (`pin`) - в самой функции можно им не пользоваться, но в сигнатуре функции он должен быть обязательно!

Сама логика выполняемого в функции кода очевидна - в консоль выводится сообщение, что `"Кнопка нажата!"`.

Отметим, что основной цикл работы программы:

```
while True:
    time.sleep(10)
```

полностью блокирует все прочие операции и в нем отсутствует фактический опрос кнопки. Фактически прерывание прерывает (тавтологии нет) сон микроконтроллера в

момент падения сигнала на пине `button` и позволяет ему "заснуть" обратно после выполнения функции `button_pressed`.

Сделаем пару дополнительных замечаний:

- Сигнатура функции обработчика прерывания определяется строго модулем `machine` языка `micropython`. При необходимости использовать дополнительные переменные они должны находиться в доступных пространствах имен.

> Если в процессе выполнения функции обработчика прерывания необходимо изменять внешние переменные следует явно определять их пространства имен через ключевые слова `global` и `nonlocal`:

```
counter = 0

def button_pressed(pin):
    global counter
    print(counter, "Кнопка нажата!")
    counter += 1
```

- Так как прерывание фактически прерывает основной цикл работы программы функция обработчик прерывания должна быть **максимально короткой и быстрой!**
- Существуют другие режимы реакции на события:

Рассмотренный ранее:

```
button.irq(trigger=Pin.IRQ_FALLING, handler=isr)
```

Срабатывание в момент изменения сигнала с низкого на высокий:

```
button.irq(trigger=Pin.IRQ_RISING, handler=isr)
```

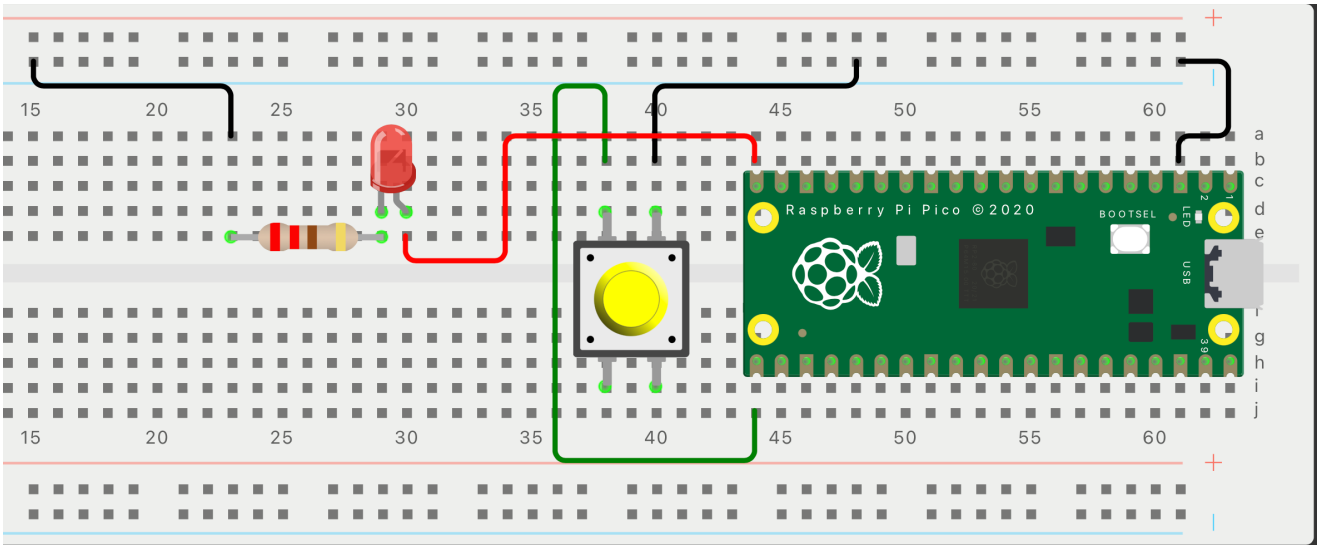
Срабатывание в момент любого изменения сигнала:

```
button.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=isr)
```

Увидеть наглядно, что прерывание само по себе не нарушает работу основной программы можно, запустив соответствующий код совместно с морганием светодиода:

```
from machine import Pin
import time

BUTTON_PIN = 16
```

```
button = Pin(BUTTON_PIN, Pin.IN, Pin.PULL_UP)

def button_pressed(pin):
    print("Кнопка нажата!")

counter = 0
def button_pressed(pin):
    global counter
    print(counter, "Кнопка нажата!")
    counter += 1

button.irq(trigger=Pin.IRQ_FALLING, handler=button_pressed)

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)
        self.led_state = None

    def on(self):
        self.led_state = "ON"
        self.led.on()

    def off(self):
        self.led_state = "OFF"
        self.led.off()

    def switch(self):
        if self.led_state == "ON":
            self.off()
        else:
            self.on()


LED_PIN = 15
led = Led(led_pin=LED_PIN)

while True:
    led.switch()
    time.sleep(2)
```



Класс кнопки с использованием прерывания

Базовое решение

Добавим логику прерывания в класс кнопки, с которого начали это занятие:

```
import time
from machine import Pin


class ButtonWithInterrupt:
    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_change_time = time.ticks_ms()
        self.pin.irq(trigger=trigger, handler=lambda pin:
self.check_button_click())

    def check_button_click(self):
        current_time = time.ticks_ms()
        if current_time - self.last_change_time > self.debounce_time:
            self.last_change_time = current_time
            print("Click!")


button = ButtonWithInterrupt(button_pin_number=16)


while True:
    time.sleep(10)
```

> В приведенном примере основной цикл программы полностью посвящен сну и отдыху микроконтроллера, очевидно, что предложенный ранее процесс моргания светодиодом также допустим к работе и опускается здесь в целях экономии места.

__Рассмотрим отличия в коде:

- В методе `__init__` добавлена строка инициализации прерывания:

```
self.pin.irq(trigger=trigger, handler=lambda pin:
self.check_button_click())
```

Отметим, что, так как функция обработчик прерывания по умолчанию принимает только объект пина, на котором сработало прерывание, для корректного вызова метода кнопки `self.check_button_click()` была использована анонимная `lambda` функция:

```
lambda pin: self.check_button_click()
```

которая принимает объект пина в качестве единственного атрибута, а внутри себя просто вызывает требуемый метод `self.check_button_click()`

- Так как прерывание срабатывает в момент нажатия кнопки (в силу триггера `trigger=Pin.IRQ_FALLING`), необходимость проверки состояния сигнала на кнопке отпадает, и в методе `check_button_click` выполняется только контроль времени с предыдущего события для устранения дребезга контактной пары кнопки.
- Отметим, что, в отличие от предыдущей версии класса кнопки, метод `self.check_button_click()` не возвращает строку `"Click!"`, а лишь выводит ее в консоль (это будет исправлено далее).

Запустив код можно отметить, что нажатие кнопки срабатывает корректно независимо от состояния загруженности микроконтроллера.

Работающая кнопка

Написанный ранее код кнопки с прерыванием корректно срабатывает, но имеет один существенный недостаток - он только выводит сообщение в консоль, а значит, напрямую использовать это для управления устройствами, как мы делали это ранее, будет нельзя.

Исправим это:

- Добавим при инициализации объекта кнопки новую переменную:

```
self.action = False
```

Атрибут `action` будет указывать, что произошло некоторое событие (нажатие кнопки).

- Изменение состояния атрибута `action` будет происходить при нажатии кнопки (взамен вывода в консоль `print("Click!")`).
- Опрос состояния переменной `action` будет выполняться в основном цикле программы и на его основании будет выполняться решение о выполнении зависящих от кнопки операций.

Собрав все это вместе, получим:

```
import time
from machine import Pin

class ButtonWithInterrupt:
    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 = time.ticks_ms()
        self.action = False
        self.pin.irq(trigger=trigger, handler=lambda pin:
self.check_button_click())

    def check_button_click(self):
        current_time = time.ticks_ms()
        if current_time - self.last_click_time > self.debounce_time:
            self.last_change_time = current_time
            self.action = True


button = ButtonWithInterrupt(button_pin_number=16, debounce_time=200)

##############################################

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)
        self.led_state = None

    def on(self):
        self.led_state = "ON"
        self.led.on()

    def off(self):
        self.led_state = "OFF"
```

```
        self.led.off()

    def switch(self):
        if self.led_state == "ON":
            self.off()
        else:
            self.on()


LED_PIN = 15
led = Led(led_pin=LED_PIN)

while True:
    if button.action:
        led.switch()
        button.action = False
```

Предложенная программа переключает при нажатии светодиод, подключенный к 15 пину.

Рассмотрев основной цикл программы:

```
while True:
    if button.action:
        led.switch()
        button.action = False
```

отметим, что:

- в предложенной версии кода необходимо явно возвращать состояния кнопки `action` в "не нажатое" положение `button.action = False`, что мы исправим в следующей версии класса;
- Если в цикле программы будет "долгий" код или задержки, нажатие кнопки будет все равно корректно обработано и не пропущено, а включение светодиода произойдет в момент проверки состояния кнопки, когда до него дойдет очередь.

> Если необходимо немедленное срабатывание, то объект кнопки должен непосредственно принимать объект, которым он управляет и менять его состояние внутри себя (реализация этого не самая простая, но и не такая сложная, как Вам может показаться - испытайте себя! Подумайте! Попробуйте!)

Улучшим класс кнопки с прерыванием!

Явным недостатком предыдущей версии класса кнопки была необходимость возврата состояния кнопки в основном цикле программы:

```
button.action = False
```

Хотя технически это не сложно, но требует от пользователя разработанного класса понимать его внутреннее устройство, а это путь к неизбежным проблемам в будущем...

Исправим это!

Как?

Да, как всегда - напишем еще один метод класса!

Рассмотрим сразу готовый код:

```
class ButtonWithInterrupt:
    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 = 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 = time.ticks_ms()
        if current_time - self.last_click_time > self.debounce_time:
            self.action = True
            self.current_action = "Click!"
            self.last_click_time = time.ticks_ms()

    def get_current_action(self):
        current_action = self.current_action
        self.current_action = None
        self.action = False
        return current_action
```

Обратим внимание на следующие изменения:

В метод `__init__` был добавлен новый атрибут `current_action`, предполагающий сохранение специфичного состояния кнопки.

При этом в методе `check_button_click` при корректном нажатии кнопки в соответствующие атрибуты объекта записываются состояния:

```
self.action = True
self.current_action = "Click!"
```

Логика в этом следующая - `action` - определяет сам факт наступления события (нажатия кнопки), а `current_action` - какое именно событие произошло (пока это справедливо может показаться избыточным, но пригодится нам при разработке кнопки обрабатывающей множественные нажатия).

Важнейшим изменением в классе является метод `get_current_action`:

```
    def get_current_action(self):
        current_action = self.current_action
        self.current_action = None
        self.action = False
        return current_action
```

Возвращающий текущее состояние событие на кнопке и сбрасывающий внутри себя состояния атрибутов `action` и `current_action`.

При этом код использования кнопки будет следующим (подключение и инициализация диода аналогично предыдущему примеру):

```
while True:
    if button.action:
        if button.get_current_action() == "Click!":
            led.switch()
```

Логика работы кода следующая:

- в основном цикле программы проверяется состояние атрибута кнопки `action` (при этом состояние атрибутов `action` и `current_action` сохраняется);
- при нажатии кнопки (`button.action = True`) в проверке условия:

```
if button.get_current_action() == "Click!":
```

вызывается метод кнопки `button.get_current_action()`, который при возвращении состояния `"Click!"` сбрасывает значения `action` и `current_action` к стартовым значениям;

- Так как условие `"Click!" == "Click!"` - верно светодиод на 15 пине меняет свое состояние.

Сделаем наш класс более "питоновским"

Сейчас наш класс корректно работает, но требует для получения состояния атрибута `current_action` выполнения метода `button.get_current_action()`.

В ООП методы возвращающие значения атрибутов называются "геттеры" (от англ. get - получать). В `python` реализация геттеров часто реализуется через специальный декоратор `@property`, позволяющий вызвать метод геттера как обычный атрибут (без скобок).

Изменим соответствующим образом наш класс:

```
class ButtonWithInterrupt:
    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 = 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 = time.ticks_ms()
        if current_time - self.last_click_time > self.debounce_time:
            self.action = True
            self._current_action = "Click!"
            self.last_click_time = time.ticks_ms()

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

При этом основной цикл программы будет выглядеть как:

```
while True:
    if button.action:
        if button.current_action == "Click!":
            led.switch()
```

Хотя логика принцип выполняемого кода не поменялись - сам код стал более "чистым" и легко читаемым.

Кнопка с "мультикликом" и прерыванием

Не останавливаясь подробно на частностях, перепишем ранее разработанный класс кнопки с "мультикликом", добавив в него логику прерываний, аналогично предыдущему примеру:

```
class Button:
    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
```

Принципиально он работает аналогично старому классу, за исключением возврата состояния кнопки через `return` в методе `check_clicks`.

Возврат соответствующих состояний выполняется аналогично разработанному классу кнопки через геттер (свойство) `current_action`.

Отметив дополнительно, что для минимизации изменений в коде класса в методе `check_button_state` оставлена проверка изменения состояния кнопки (с нажатой на отпущенную и наоборот) аналогично предыдущий версии класса. Для сохранения работоспособности этого метода прерывание пина кнопки было настроено в соответствующий режим:

```
trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING
```

Таймеры

Таймеры в микроконтроллерах — это важный инструмент для работы с временными интервалами. Они позволяют выполнять задачи с точной задержкой, периодически вызывать функции, измерять время и многое другое. В MicroPython на Raspberry Pi Pico (RP2040) таймеры предоставляются через модуль `machine.Timer`.

Что такое таймер?

Таймер — это аппаратный или программный счетчик, который отсчитывает время в заданных единицах (например, миллисекунды или микросекунды). Таймеры могут работать в двух основных режимах:

- **Однократный режим (`ONE_SHOT`)**: Таймер срабатывает один раз через заданный интервал времени.
- **Периодический режим (`PERIODIC`)**: Таймер срабатывает периодически через заданный интервал времени.

2. Таймеры в MicroPython

В MicroPython таймеры доступны через класс `machine.Timer`. На Raspberry Pi Pico (RP2040) поддерживаются как аппаратные, так и виртуальные таймеры.

Основные методы и параметры:

- **`Timer.init()`**: Инициализация таймера.
  - `mode`: Режим работы (`Timer.ONE_SHOT` или `Timer.PERIODIC`).
  - `period`: Интервал времени в миллисекундах.
  - `callback`: Функция, которая будет вызвана при срабатывании таймера.
- **`Timer.deinit()`**: Остановка и выключение таймера.

ONE_SHOT таймер

Рассмотрим пример использования однократного таймера:

```
from machine import Timer
from time import ticks_ms


def timer_callback(timer):
    print(ticks_ms(), "Таймер сработал!")

# Создаем таймер
timer = Timer()

# Настраиваем таймер на однократное срабатывание через 2000 мс (2 секунды)
timer.init(mode=Timer.ONE_SHOT, period=2000, callback=timer_callback)
print(ticks_ms())

while True:
    pass
```

- Для работы с таймером необходимо создать соответствующий объект таймера:

```
timer = Timer()
```

Пока созданный таймер не работает и бесполезен. Его инициализация выполняется через его метод `init`:

```
timer.init(mode=Timer.ONE_SHOT, period=2000, callback=timer_callback)
```

После инициализации таймера ровно по прошествии 2000 миллисекунд (2 секунд) будет вызвана функция `timer_callback`:

```
def timer_callback(timer):
    print(ticks_ms(), "Таймер сработал!")
```

> Отметим, что аналогично функции обработчика прерывания, функция, вызываемая по таймеру, обязательно должна принимать в себя сам объект таймера!

Дополнительно отметим, что к моменту срабатывания таймера, микроконтроллер уже будет безнадежно зациклен в конструкции:

```
while True:
    pass
```

И сообщение `print(ticks_ms(), "Таймер сработал!")` из функции `timer_callback` будет его последними словами (до момента перезагрузки или изменения кода программы).

PERIODIC таймер

Похожим образом работает периодический таймер, за тем лишь исключением, что он будет автоматически перезапускаться после окончания каждого из своих циклов:

```
from machine import Timer
import time


# Функция, которая будет вызвана при срабатывании таймера
def timer_callback(timer):
    print(time.ticks_ms(), "Таймер сработал!")

# Создаем таймер
timer = Timer()

# Настраиваем таймер на периодическое срабатывание каждые 500 мс
timer.init(mode=Timer.PERIODIC, period=500, callback=timer_callback)

# Ждем 3 секунды
time.sleep(3)
# Останавливаем таймер
timer.deinit()
time.sleep(0.001)
print("Таймер остановлен.")

while True:
    pass
```

Создаваемый таймер будет вызывать каждые 500 миллисекунд функцию `timer_callback`, несмотря на 3 секундный "сон" микроконтроллера `time.sleep(3)` после чего таймер "выключается" (деинициалируется):

```
timer.deinit()
```

> Отметим, что после "деинициализации" таймер может быть заново "инициализирован" в любом режиме.

"Моргание" светодиодом с помощью таймера

```
from machine import Timer, Pin

# Настройка светодиода
LED_PIN = 15
led = Pin(LED_PIN, Pin.OUT)

# Функция, которая будет вызвана при срабатывании таймера
def toggle_led(timer):
    led.toggle()  # Переключаем состояние светодиода

# Создание и настройка таймера
timer = Timer(-1)  # -1 означает виртуальный таймер
timer.init(mode=Timer.PERIODIC, period=500, callback=toggle_led)  # 500 мс
= 0.5 секунды

# Основной цикл программы
while True:
    pass
```

Отметим, что в таком режиме точность периодичности включения и выключения светодиода будет находиться в пределах 1 миллисекунды, независимо от загруженности микроконтроллера прочими задачами.

Кнопка с "мультикликом", прерыванием и таймером

Разработанный класс кнопки с мультикликом и прерыванием корректно работает.

В большинстве случаев на этом стоило бы и остановиться, не усложняя сущность без необходимости, не исправляя то, что не сломано и не тратя ценные ресурсы на не обязательные при решении проблемы сущности (таймеры).

Однако, для углубления понимания темы таймеров используем их в кнопке с "мультикликом".

Без лишних предисловий рассмотрим готовый код:

```
class Button:
    def __init__(self, pin_num, debounce_time=50, click_timeout=500):
        self.button = Pin(pin_num, Pin.IN, Pin.PULL_UP)
        self.debounce_time = debounce_time
        self.click_timeout = click_timeout
        self.click_count = 0
        self.last_press_time = 0
        self.timer = Timer()
        self._click_event = None
        self.action = False
        self.button.irq(trigger=Pin.IRQ_FALLING,
handler=self.button_pressed)

    def button_pressed(self, pin):
        current_time = time.ticks_ms()
        if time.ticks_diff(current_time, self.last_press_time) >
self.debounce_time:
            self.click_count += 1
            self.last_press_time = current_time
            # Перезапускаем таймер для ожидания следующего клика
            self.timer.deinit()
            self.timer.init(mode=Timer.ONE_SHOT,
period=self.click_timeout, callback=self.handle_clicks)

    def handle_clicks(self, timer=None):
        if self.action is False:
            self.action = True
            if self.click_count == 1:
                self._click_event = "single"
            elif self.click_count == 2:
                self._click_event = "double"
            elif self.click_count == 3:
                self._click_event = "triple"
            else:
                self._click_event = f"multiple - {self.click_count}"
            self.click_count = 0

    @property
    def click_event(self):
        click_event = self._click_event
        self._click_event = None
        self.action = False
        return click_event
```

Первое, что бросается в глаза это "упрощение" и большая "чистота" кода, достигаемая за счет исключения проверок состояния кнопки.

Фактически принцип работы кнопки заключается в следующем:

Прерывание на кнопке запускает метод `button_pressed`, который отвечает за подсчет количества нажатий кнопки, изменяя количество кликов в атрибуте `click_count`, и после каждого корректного нажатия перезапускающий однократный таймер на время `click_timeout` (определяющее полное допустимое время между кликами в одной команде).

Если время между кликами превышает `click_timeout`, таймер срабатывает и запускает метод `handle_clicks`, определяющий логику инициации атрибутов `action` и `_click_event` соответствующими значениями.

> Обратите внимание на то, как можно избежать необходимости использования `lambda` функций в обработчиках прерываний и таймеров (добавляя в сигнатуру самих методов соответствующие аргументы)!

Пример использования кнопки в коде:

```
button = Button(pin_num=16, debounce_time=200, click_timeout=500)

while True:
    if button.action:
        ce = button.click_event
        if ce == "single":
            print("Одинарный клик!")
        elif ce == "double":
            print("Двойной клик!")
        elif ce == "triple":
            print("Тройной клик!")
        else:
            print(ce)
    time.sleep(0.1)
```

или

```
button = Button(pin_num=16, debounce_time=200, click_timeout=500)

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)
        self.led_state = None

    def on(self):
        self.led_state = "ON"
        self.led.on()

    def off(self):
        self.led_state = "OFF"
        self.led.off()
```

```
    def switch(self):
        if self.led_state == "ON":
            self.off()
        else:
            self.on()

LED_PIN = 15
led = Led(led_pin=LED_PIN)

while True:
    ce = button.click_event
    if ce == "single":
        led.on()
    if ce == "double":
        led.off()
    if ce == "triple":
        led.switch()
    time.sleep(1)
```