

Inhaltsverzeichnis

Grundsätzliches	2
Die Dateien	2
Das Funktionsprinzip	2
Die Hardware	3
Raspberry Pi Pico W	3
Pico Explorer Base von Pimoroni	3
Sonstiges	3
Pinbelegung	3
Die Entwicklungsumgebung	4
Installation von Micropython	4
Hello World!	5
Die blinkende Led	5
Zusammenfassung	5
Eine zweite Led	5
Zusammenfassung	6
Das Nachrichtensystem	7
Der Empfänger	7
Ein Taster als Sender	7
Das Schweigen der Led	7
Bye Bye	8
Eine Funktion zum Abschied (verzögerte Funktion)	8
Die einfache Lösung	8
Der Zähler (wiederholte Funktion)	9
Zusammenfassung	10
Anhang	11
Weitere Unterlagen	11
Lizenz	11

Grundsätzliches

Diese Micropython Multitasking Framework (MMF) ist ein experimenteller Code, der nur Tests zur Machbarkeit eines solchen Frameworks erlauben soll.

Programme in Python sind sehr schlecht für Echtzeitanwendungen geeignet. Oft ist das aber auch nicht notwendig. Es genügt, wenn gewisse Vorgänge scheinbar selbstständig im Hintergrund ablaufen.

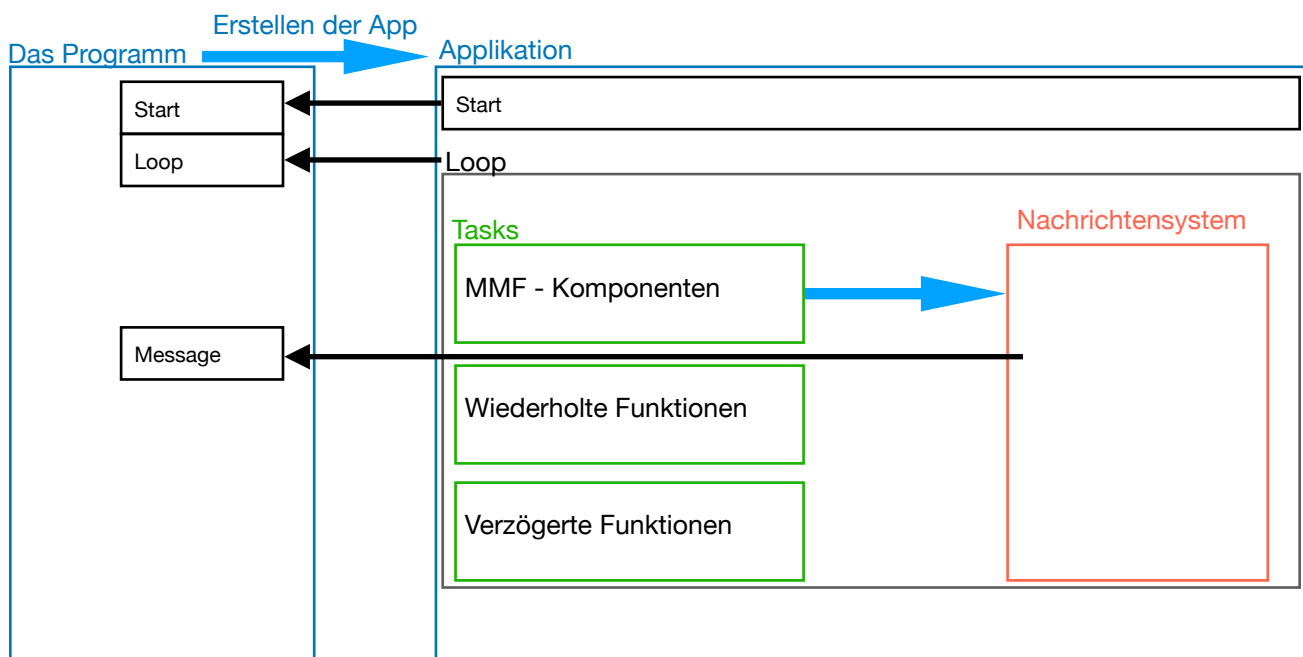
Dazu bietet Micropython bereits einige Hilfen mit **asyncio** und **_thread**. Doch diese weisen für Einsteiger einige Tücken auf. **MMF** soll für Einsteiger einfache und nachvollziehbare Funktionen zur Verfügung stellen. Dies erfolgt durch Standardkomponenten, die der Benutzer einfach verwenden kann.

Für Fortgeschrittene soll es aber trotzdem möglich sein, das System beliebig zu erweitern.

Die Dateien

Alle Dateien des Frameworks werden in ein Unterverzeichnis **lib** auf den Microcontroller kopiert. Immer dabei sind die hardwareunabhängigen Dateien **MMFClasses.py** und **MMFComponents.py**. Da Micropython je nach Microcontroller verschiedene Implementierungen aufweist, gibt es auch noch ein vom Microcontroller abhängiges Modul. Hier in unserem Fall ist das **MMF_RP2040.py**. Manchmal enthält das Board zusätzliche Hardware oder es wird ein Extensionboard verwendet. In unserem Fall ist es das Pico Explorer - Board. Deshalb verwenden wir noch das Modul **MMF_Explorer.py**.

Das Funktionsprinzip



Das Hauptprogramm erstellt die Applikation. Es legt Variablen an, fügt die gewünschten Tasks hinzu und startet zum Schluss die Endlosschleife der Applikation. Ab diesem Zeitpunkt agiert die App selbstständig, kann aber mit seinem Umfeld kommunizieren. Das Hauptprogramm hat verschiedene Einflussmöglichkeiten.

In der Startphase und bei jedem Schleifendurchgang kann eine Funktion des Hauptprogramms aufgerufen werden.

Verschiedene Komponenten senden automatisch Informationen zu ihrem Status an das interne Nachrichtensystem. Das Hauptprogramm hat die Möglichkeit, all diese Nachrichten mitzulesen und entsprechend darauf zu reagieren.

Das erscheint auf den ersten Blick kompliziert. In der Praxis ist es aber sehr einfach.

Die Hardware

Raspberry Pi Pico W

Bei den Experimenten wird ein **Raspberry Pi Pico W** verwendet. Der WiFi - Teil des Boards wird aber nicht verwendet, so dass ein **Raspberry Pi Pico** ebenfalls verwendet werden kann.

<https://www.raspberrypi.com/products/raspberry-pi-pico/>

Pico Explorer Base von Pimoroni

Für die Experimente wurde der Pico noch um dieses Board erweitert. Dadurch stehen unter anderem ein Display und 4 Buttons zur Verfügung.

<https://www.raspberrypi.com/products/raspberry-pi-pico/>

Sonstiges

Zusätzlich wurden noch 3 farbige Leds und ein Potentiometer angeschlossen.

Pinbelegung

GPIO (General Purpose Input Output)

Diese Anschlüsse können als digitale Ein- und Ausgänge konfiguriert werden.

Alle Ausgänge sind PWM fähig und haben bereits einen 100 Ohm Widerstand in Serie eingebaut.

GP0 = 0 (rot)
GP1 = 1 (gelb)
GP2 = 2 (grün)
GP3 = 3 (Buzzer)
GP4 = 4 (PIR)
GP5 = 5
GP6 = 6
GP7 = 7

Taster

Die Taster belegen die GPIOs 12 - 15 und verbinden zu GND.

Button A = 12
Button B = 13
Button X = 14
Button Y = 15

Motoren (nicht wirklich zu gebrauchen, da keine externe Stromversorgung vorgesehen ist!)

Die Motortreiber belegen die GPIOs 8 - 11.

Wir werden sie in den Experimenten nicht verwenden.

Motor 1 (-) = 8
Motor 1 (+) = 9
Motor 2 (-) = 10
Motor 2 (+) = 11

SPI

SPI wird für das Interne Display verwendet.

SPI MISO - 16
LCD CS - 17
SPI SCK - 18
SPI MOSI - 19

I2C

I2C SDA - 20

I2C SCL - 21

I2C INT - 22

Analoge Eingänge

ADC0 = 26 (Potentiometer)

ADC1 = 27

ADC2 = 28

Das Schemahttps://cdn.shopify.com/s/files/1/0174/1800/files/pico_explorer_schematic.pdf?v=1619077703**Der Pico**<https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html>**Die Entwicklungsumgebung**

Wir verwenden Thonny (<https://thonny.org/>). Dieser Editor unterstützt eine grosse Palette von Pythonversionen und Boards. Wir wählen unter **Tools / Options / Interpreter** die Option **Micropython (Raspberry Pi Pico)** aus.

Installation von Micropython

Die Installation kann direkt in Thonny erfolgen. Wir wählen unter **Tools / Options / Interpreter** rechts unten **Install or update MicroPython** aus.
Sobald das Target Volume richtig erkannt ist, kann das Pimoroni - Board ausgewählt werden (**Raspberry Pi Pico / Pico H (with Pimoroni libraries)**).

Der Ordner **lib** mit den Dateien

MMFClasses.py, MMFComponents.py, MMF_RP2040.py und MMF_Explorer.py
muss auf das Board kopiert werden.

Hello World!

Das typische Hello World - Programm bei Microcontrollern ist eine blinkende Led.

Die blinkende Led

Wir importieren gleich die ganze Unterstützung für das Explorerboard.

```
from MMF_Explorer import *
```

Zuerst erzeugen wir eine leere App.

```
app = Application()
```

Wir wollen mit einer Led arbeiten, daher erzeugen wir eine Led - Komponente. Eine Led ist an einen digitalen Ausgang angeschlossen, daher benutzen wir eine rote Led an Ausgang GP0.

```
rot = DigitalOut(0)
```

Diese Komponente fügen wir der App hinzu.

```
app.add_components(rot)
```

Die Led soll mit einer Frequenz von 1 Hz blinken.

```
rot.blink = 1
```

Jetzt übergeben wir die Kontrolle der App und starten diese.

```
app.run()
```

Damit ist die Aufgabe bereits erfüllt. Wie du siehst, muss man nicht immer alle Möglichkeiten nutzen, die das Framework bereithält.

Zusammenfassung

```
from MMF_Explorer import *
```

```
app = Application()
```

```
rot = DigitalOut(0)
```

```
app.add_components(rot)
```

```
rot.blink = 1
```

```
app.run()
```

Das ist jetzt nicht besonders beeindruckend. Das hätte man ohne Framework ebenfalls hingekriegt.

Eine zweite Led

Die zweite Led (grün an GP2) soll jetzt mit einer Frequenz von 2 Hz blinken. Mit dem Framework kein Problem!

Wir erzeugen die zweite Led, fügen sie der Applikation hinzu und setzen ihre Frequenz auf 2 Hz.

```
gruen = DigitalOut(2)
```

```
app.add_components(rot, gruen)
```

```
gruen.blink = 2
```

Damit ist auch schon die Abfolge bei so einfachen Programmen klar.

- Importe
- App erzeugen
- MMF - Komponenten erzeugen
- MMF - Komponenten vorbereiten
- App starten

Zusammenfassung

```
# Importe
from MMF_Explorer import *

# Applikation erzeugen
app = Application()

# MMF - Komponenten erzeugen, hinzufügen und konfigurieren
rot = DigitalOut(0)
gruen = DigitalOut(2)

app.add_components(rot, gruen)

rot.blink = 1
gruen.blink = 2

# App starten
app.run()
```

Das hättest du ohne Framework kaum mit so wenig Code realisieren können.

Das Nachrichtensystem

Wir speichern das aktuelle Hello World - Projekt unter dem Namen **Nachrichten.py** ab.

Der Empfänger

Wir benötigen eine Funktion, die die Nachrichten entgegennimmt. Um zu schauen, was da so kommt, geben wir die Informationen auf der Konsole aus.

```
def on_message(sender, topic, data):
    print(sender, topic, data)
```

Jetzt muss die Funktion noch bei der App angemeldet werden. Dazu ändern wir den Startbefehl.

```
app.run(message=on_message)
```

Ein Taster als Sender

Die Standardkomponente Button gibt uns Informationen über den Zustand eines Tasters.

In den Unterlagen zum Explorer - Board sehen wir, dass Button A an GP12 angeschlossen ist. Wir erzeugen die Komponente und fügen sie der App hinzu.

```
button_a = Button(12)
app.add_components(rot, gruen, button_a)
```

Damit erhalten wir bereits Nachrichten vom Taster A. Um das besser sehen zu können, geben wir nur die Nachrichten des Tasters aus.

```
def on_message(sender, topic, data):
    if sender == button_a:
        print(sender, topic, data)
```

Beim Druck auf die Taste bekommen wir zwei Nachrichten:

'**changed**' sagt, dass sich etwas geändert hat und mit **True** wird signalisiert, dass jetzt der Taster gedrückt ist.

'**pressed**' sagt, dass der Taster gedrückt wurde. Der Wert in **data** ist nicht relevant.

Beim Loslassen erhalten wir ebenfalls zwei Meldungen:

'**changed**' sagt, dass sich etwas geändert hat und mit **False** wird signalisiert, dass der Taster jetzt nicht mehr gedrückt ist.

'**released**' sagt, dass der Taster losgelassen wurde. In **data** wird angegeben, wie lange der Taster gedrückt war (in ms).

Das Schweigen der Led

Jetzt wollen wir durch Druck auf den Taster die grüne Led zum Schweigen bringen. Sobald der Taster losgelassen wird, darf die Led wieder blinken.

```
def on_message(sender, topic, data):
    if sender == button_a:
        if topic == 'pressed':
            gruen.high = False
        elif topic == 'released':
            gruen.blink = 2
```

Bye Bye

Jetzt verabschieden wir uns. Natürlich ist damit nicht dieses Tutorial zu Ende! Wir versuchen nur, das Programm auf anständige Art zu beenden.

Eine Funktion zum Abschied (verzögerte Funktion)

Diesmal verwenden wir eine Komponente, die zum Explorer - Board gehört: das Display.

Durch einen Druck auf Taste B soll sich unser Programm mit dem Text **Bye** verabschieden und sich danach selbst beenden.

Die einfache Lösung

Zuerst benötigen wir zwei zusätzliche Komponenten: den Taster B und das Display.

Beginnen wir mit dem Display. Das ist keine MMF - Komponente und darf der Applikation nicht hinzugefügt werden. Daher erstellen wir es gleich nach der App.

```
display = Display()
```

Auf bei den MMF - Komponenten gibt es eine Änderung. Für Button A mussten wir nachschauen, an welchem Port der Taster angeschlossen ist. Das geht bequemer. Im MMF_Explorer Modul sind die Taster bereits vordefiniert!

```
button_a = ButtonA()
button_b = ButtonB()
```

Beide sind normale MMF - Komponenten und müssen der Applikation hinzugefügt werden.

```
app.add_components(rot, gruen, button_a, button_b)
```

Wir erstellen zuerst eine Hilfsfunktion, die das Display löscht und einen Text an eine bestimmte Position schreibt.

```
def print_text(txt, x, y, scale):
    display.clear_all()
    display.print(txt, x, y, scale)
    display.update()
```

Jetzt benötigen wir noch eine Funktion, die die Verabschiedung durchführt.

```
def byebye():
    print_text("Bye", 40, 80, scale=10)
    app.stop()
```

Wir schreiben "Bye" auf das Display und beenden das Programm.

byebye() muss noch aufgerufen werden. Das passiert, wenn Taste B gedrückt wird.

```
def on_message(sender, topic, data):
    if sender == button_a:
        ...
    elif sender == button_b:
        if topic == 'pressed':
            byebye()
```


Eine Verbesserung mit verzögerter Funktionen

Die bisherige Lösung ist nicht besonders schön. Der Text bleibt stehen und die Leds leuchten weiter, obwohl das Programm zu Ende ist. Wir sollten das verbessern:

- Es wird "Bye" auf das Display geschrieben.
- Nach 5 Sekunden sollen die Leds das Blinken beenden und sich ausschalten. Das Display soll dunkel werden und das Programm wird beendet.

Dazu erstellen wir eine neue Funktionen.

aus() schaltet alle Leds aus, löscht das Display und beendet die App.

```
def aus():
    rot.high = False
    gruen.high = False
    display.clear_all()
    display.update()
    app.stop()
```

Diese Funktion rufen wir jetzt zeitverzögert auf.

```
def byebye():
    print_text("Bye", 40, 80, scale=10)
    app.after(5000, aus)
```

Das Programm läuft immer noch gleich schnell durch **byebye(). print_text()** wird sofort aufgerufen. Dann wird der Auftrag erteilt, in 5000 ms den Stopp des Programms auszulösen.

Der after - Aufruf enthält immer zuerst die Zeitverzögerung in Millisekunden (3000). Danach kommt der Funktionsname ohne Argumente und Klammern (app.stop). Wenn Argumente übergeben werden müssen, können diese anschliessend angegeben werden (, 10, 80, 6).

Der Zähler (wiederholte Funktion)

Wir kopieren das aktuelle Projekt in eine Datei mit dem Namen **Zaehler.py**. In diesem Beispiel erstellen wir eine zeitgesteuerte Funktion.

Eine Funktion soll einen Wert im Sekundentakt hochzählen und auf dem Display anzeigen. Dazu benötigen wir zuerst eine Zählvariable. Diese legen wir als Eigenschaft der App an.

```
app.zaehlerstand = 0
```

Dann benötigen wir noch die Zählfunktion. Das Resultat wird direkt auf den Bildschirm geschrieben.

```
def zaehlen():
    app.zaehlerstand += 1
    print_text(str(app.zaehlerstand), 80, 80, 10)
```

Diese kann der App hinzugefügt werden.

```
app.zaehler = app.add_function(1000, zaehlen)
```

Beim Beenden der App soll auch der Zähler gestoppt werden.

```
def byby():
    app.zaehler.stop()
    ...
```

Zusammenfassung

```
# Importe
from MMF_Explorer import *

# Applikation erzeugen
app = Application()
app.zaehlerstand = 0
display = Display()

# MMF - Komponenten erzeugen, hinzufügen und konfigurieren
rot = DigitalOut(0)
gruen = DigitalOut(2)
button_a = ButtonA()
button_b = ButtonB()

app.add_components(rot, gruen, button_a, button_b)

rot.blink = 1
gruen.blink = 2

# Funktionen
def print_text(txt, x, y, scale):
    display.clear_all()
    display.print(txt, x, y, scale)
    display.update()

def aus():
    rot.high = False
    gruen.high = False
    display.clear_all()
    display.update()
    app.stop()

def byebye():
    app.zaehler.stop()
    print_text("Bye", 40, 80, scale=10)
    app.after(5000, aus)

def zaehlen():
    app.zaehlerstand += 1
    print_text(str(app.zaehlerstand), 80, 80, 10)

app.zaehler = app.add_function(1000, zaehlen)

# Nachrichtenempfänger
def on_message(sender, topic, data):
    if sender == button_a:
        if topic == 'pressed':
            gruen.high = False
        elif topic == 'released':
            gruen.blink = 2
    if sender == button_b:
        if topic == 'pressed':
            byebye()

# App starten
app.run(message=on_message)
```

Anhang

Weitere Unterlagen

Das Programmierhandbuch für Einsteiger (***MMFProgrammierhandbuchEinsteiger***) sollte als Begleitmaterial für dieses Tutorial verwendet werden.

Für eine weitere Vertiefung wird das Komponententutorial (***MMFKomponententutorial***) empfohlen.

Lizenz

Dieses Dokument und der darin erhaltene Code untersteht einer MIT Lizenz und ist daher frei verfügbar. Die beschriebene Hardware, die verwendeten Tools und die Informationen auf den verlinkten Seiten unterstehen aber den Urheber- und Markenrechten der betreffenden Hersteller.