# Inhaltsverzeichnis

# Grundsätzliches

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

Dies ist das zweite Tutorial. Als Grundlage solltest du zuerst das ***Einführungstutorial*** durcharbeiten. Als Begleitmaterial sollte das ***Programmierhandbuch für Einsteiger*** immer griffbereit sein.

## 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 (Led rot)
GP1 = 1 (Led gelb)
GP2 = 2 (Led grün)
GP3 = 3 (Buzzer)
GP4 = 4 (PIR)

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

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

**Analoge Eingänge**
ADC0 = 26 (Potentiometer)
ADC1 = 27
ADC2 = 28

**Das Schema**
https://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 Applikation

Die Applikation ist keine Komponente, ist aber doch ein wichtiger Bestandteil des Frameworks. Sie ist verantwortlich für die Ausführung aller Aufgaben (Tasks), die wir ihr übertragen.

## Der Programmaufbau (Applikation1.py)

Der grundsätzliche Aufbau eines Programms ist immer derselbe:

Import des MMF - Frameworks. Dabei wird immer das hardwareabhängige Modul importiert.

```
from MMF_RP2040 import *
```

Dann wird eine App erzeugt.

```
app = Application()
```

Dieser App werden verschiedene Aufträge erteilt. Zum Schluss wird sie gestartet, damit sie die Aufträge ausführen kann.

```
app.run()
```

## Die Hilfsfunktion millis() (Applikation1.py)

Wir benötigen einen Zeitstempel in Millisekunden, der laufend erhöht wird.
Die Funktion .millis() liefert die Anzahl Millisekunden seit Erzeugung der App zurück.

```
from MMF_RP2040 import *
import time

app = Application()

print(app.millis())
time.sleep(0.01)
print(app.millis())
```

Wir erhalten folgende Ausgabe:

```
3
13
```

Es sind also 10 ms vergangen, was exakt der Sleepzeit von 0.01 Sekunden entspricht.
app.run() musste nicht aufgerufen werden, da noch keine Aufgaben vorhanden sind.

## Funktionen (Application2.py)

Die Applikation kann gewöhnliche Funktionen in Tasks zu verwandeln. Dabei wird die Aufgabe dann automatisch der Applikation zugewiesen. Diese Auftragserteilung erfolgt nahezu verzögerungsfrei. Die Aufgaben werden dann automatisch zur korrekten Zeit ausgeführt.

**Wiederholte Funktionen**

Wir schreiben eine gewöhnliche Funktion. Jetzt möchten wir, dass diese Funktion alle 500 ms ausgeführt wir.

Als Beispiel wollen wir einen Zähler realisieren, der von 0 aus hochzählt. Dazu benötigen wir eine Zählvariable. In Python haben globale Variablen diverse Tücken. Deshalb erstellen wir die Zählvariable ails Bestandteil der App.

```
app = Application()
app.zaehlerstand = 0
```

Der Zähler wird als Funktion realisiert.

```
def zaehle():
    app.zaehlerstand += 1
    print(f'Stand {app.zaehlerstand} nach {app.zaehlerstand*0.5} Sekunden')
```

Diese Funktion muss jetzt alle 500 Millisekunden aufgerufen werden.

```
app.zaehler = app.add_function(500, zaehle)
```

Damit haben wir unseren ersten Task erstellt. Oft möchte man solche Aufgaben später noch beeinflussen. Darum wird der Task als Eigenschaft der App gespeichert (***app.zaehler***).

***add_function()*** erstellt aus einer Funktion eine Aufgabe und fügt sie in die App ein.
Das erste Argument gibt immer das ***Zeitintervall*** in Millisekunden an, in dem die Funktion aufgerufen werden soll.
Danach wird der ***Funktionsname*** angegeben. Es darf nur der Name angegeben werden. Die beiden Klammern dürfen hier nicht angegeben werden.
Zum Schluss können noch optional ***Argumente*** angegeben werden. Da ***zaehle()*** keine Argumente hat, werden wir das erst später an anderer Stelle verwenden.

```
from MMF_RP2040 import *

app = Application()
app.zaehlerstand = 0

def zaehle():
    app.zaehlerstand += 1
    print(f'Stand {app.zaehlerstand} nach {app.zaehlerstand*0.5} Sekunden')

app.zaehler = app.add_function(500, zaehle)

print('Das Programm wird gestartet')
app.run()
print('Das Programm ist beendet')
```

Die Ausgabe:

```
Das Programm wird gestartet
Stand 1 nach 0.5 Sekunden
Stand 2 nach 1.0 Sekunden
Stand 3 nach 1.5 Sekunden
Stand 4 nach 2.0 Sekunden
Stand 5 nach 2.5 Sekunden
Stand 6 nach 3.0 Sekunden
Stand 7 nach 3.5 Sekunden
...
```

Das Programm wird gestartet und die Aufgabe wie gewünscht ausgeführt. ***app.run()*** ist eine Endlosschleife. Daher wird das Programm nie beendet.

**Verzögerte Funktionen**
Wir schreiben eine gewöhnliche Funktion zum Beenden des Programms. Jetzt möchten wir, dass diese Funktion nach 6 Sekunden einmalig ausgeführt wird.

```
def programmende(text):
    print(text)
    app.stop()
```

Diese Funktion schreibt einen Text in die Konsole und beendet dann das Programm. Wir müssen sie also nur noch der App als Task übergeben.

```
app.after(6000, programmende, 'Bei Sekunde 6 wird das Programm beendet.')
```

Damit haben wir einen weiteren Task erstellt. Da dieser nur einmalig ausgeführt wird und sich danach selbst entfernt, lohnt es sich nicht, ihn in eine Variable zu speichern.

Das erste Argument gibt immer das ***Zeitverzögerung*** in Millisekunden an, nach der die Funktion aufgerufen werden soll.
Danach wird der ***Funktionsname*** angegeben. Es darf nur der Name angegeben werden. Die beiden Klammern dürfen hier nicht angegeben werden.
Hier haben wir das ***Argument text***, das hinter dem Funktionsnamen angegeben werden kann. Mehrere Argumente werden einfach mit Komma getrennt aufgeführt. Es sind auch benannte Argumente möglich.

```
from MMF_RP2040 import *

app = Application()
app.zaehlerstand = 0

def programmende(text):
    print(text)
    app.stop()

def zaehle():
    app.zaehlerstand += 1
    print(f'Stand {app.zaehlerstand} nach {app.zaehlerstand*0.5} Sekunden')

app.zaehler = app.add_function(500, zaehle)
app.after(6000, programmende, 'Bei Sekunde 6 wird das Programm beendet.')

print('Das Programm wird gestartet')
app.run()
print('Das Programm ist beendet')
```

Die Ausgabe:

```
Das Programm wird gestartet
Stand 1 nach 0.5 Sekunden
Stand 2 nach 1.0 Sekunden
Stand 3 nach 1.5 Sekunden
Stand 4 nach 2.0 Sekunden
Stand 5 nach 2.5 Sekunden
Stand 6 nach 3.0 Sekunden
Stand 7 nach 3.5 Sekunden
Stand 8 nach 4.0 Sekunden
Stand 9 nach 4.5 Sekunden
Stand 10 nach 5.0 Sekunden
Stand 11 nach 5.5 Sekunden
Stand 12 nach 6.0 Sekunden
Bei Sekunde 6 wird das Programm beendet.
Das Programm ist beendet
```

**Ein weiteres Beispiel**

Hier noch eine Erweiterung zum Selbststudium:

```
from MMF_RP2040 import *

app = Application()
app.zaehlerstand = 0

def schreibe_text(text):
    print(text)

def programmende(text):
    print(text)
    app.stop()

def stoppe_zaehler(text):
    print(text)
    app.zaehler.stop()

def zaehle():
    app.zaehlerstand += 1
    print(f'Stand {app.zaehlerstand} nach {app.zaehlerstand*0.5} Sekunden')

app.zaehler = app.add_function(500, zaehle)
app.after(6000, programmende, 'Bei Sekunde 6 wird das Programm beendet.')
app.after(4000, stoppe_zaehler, 'Nach 4 Sekunden wird der Zähler gestoppt.')
app.after(3000, schreibe_text, 'Sekunde 3')
app.after(2000, schreibe_text, 'Sekunde 2')

print('Das Programm wird gestartet')
app.run()
print('Das Programm ist beendet')
```

## Das Nachrichtensystem (Application3.py)

Die Tasks können Informationen enthalten, die wir gerne in unserem Programm verarbeiten möchten. Das selbst dauernd abzufragen wäre etwas mühsam. Daher versenden viele Komponenten Informationen, wenn sich etwas ändert.

### Der Nachrichtenempfänger

Unsere Tasks sind zwar keine vollwertigen Komponenten, können aber trotzdem Nachrichten versenden. Doch zuerst müssen wir einen Nachrichtenempfänger erstellen und ihn bei der App anmelden.

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

app.run(message=on_message)
```

Wenn wir das Programm starten, werden wir feststellen, dass noch keine Nachrichten verschickt werden. Wir können die Zählfunktion so ausbauen, dass sie bei jeder Zählererhöhung eine Nachricht verschickt.

```
def zaehle():
    app.zaehlerstand += 1
    text = (f'Stand {app.zaehlerstand} nach {app.zaehlerstand*0.5} Sekunden')
    app.notify(app.zaehler, 'count', text)
```

Wir erhalten jetzt den Zählerstand als Nachricht.

Eine Nachricht übermittelt immer folgende Informationen:
***sender*** ist der Absender der Nachricht. Normalerweise ist das ein Task.
Im ***topic*** steht, um was es sich bei der Nachricht handelt.
***data*** übermittelt weitere Details.

### Ein anderes Zählprogramm

```
from MMF_RP2040 import *

app = Application()
app.zaehlerstand = [0, 0]

def zaehle(index, faktor):
    app.zaehlerstand[index] += 1
    stand = app.zaehlerstand[index]
    text = f'Zähler {index}: {stand} nach {stand*faktor} Sekunden'
    app.notify(app.zaehler[index], 'count', text)

app.zaehler = [
    app.add_function(500, zaehle, 0, 0.5),
    app.add_function(1000, zaehle, 1, 1)]

app.after(6000, app.stop)
app.after(3000, app.zaehler[0].stop)

def on_message(sender, topic, data):
    if sender == app.zaehler[1]:
        print('   ', sender, topic, data)
    else:
        print(sender, topic, data)

app.run(message=on_message)
```

Jetzt benutzen wir zwei Zähler und lassen uns durch sie informieren. Für die beiden Zähler und ihre Zählvariablen benutzen wir eine Liste.

```
app.zaehlerstand = [0, 0]

app.zaehler = [
    app.add_function(500, zaehle, 0, 0.5),
    app.add_function(1000, zaehle, 1, 1)]
```

Die ***Zählfunktion*** kann mit beiden Zählern umgehen. Darum muss sie angepasst werden.

```
def zaehle(index, faktor):
    app.zaehlerstand[index] += 1
    stand = app.zaehlerstand[index]
    text = f'Zähler {index}: {stand} nach {stand*faktor} Sekunden'
    app.notify(app.zaehler[index], 'count', text)
```

Welcher ***Index*** und welcher ***Faktor*** verwendet wird, wird in ***add_function()*** angegeben.

```
app.zaehler = [
    app.add_function(500, zaehle, 0, 0.5),
    app.add_function(1000, zaehle, 1, 1)]
```

In ***on_message()*** können wir jetzt anhand des Senders feststellen, welcher Zähler die Nachricht schickt. Damit lässt sich eine Einrückung bei der Ausgabe realisieren.

```
def on_message(sender, topic, data):
    if sender == app.zaehler[1]:
        print('   ', sender, topic, data)
    else:
        print(sender, topic, data)
```

# Digitale Ausgänge

DigitalOut ist der einfachste Ausgang. Der Ausgangspin kann die Zustände LOW (0 V) oder HIGH (Betriebsspannung, hier 3.3 V) annehmen.

PWMOut ist von DigitalOut abgeleitet, kann aber anstelle eines HIGH - Signals ein Rechtecksignal mit einstellbarer Pulsbreite ausgeben.

## DigitalOut (DigitalOut.py)

- **Wir erstellen ein Programm mit einer roten Led am digitalen Ausgang 0 und einer grünen Led am digitalen Ausgang 2.**
- **Die rote Led soll dunkel bleiben, die grüne Led soll leuchten.**
- **Alle Nachrichten werden auf der Konsole ausgegeben.**

```
from MMF_RP2040 import *

app = Application()

rot = DigitalOut(0)
gruen = DigitalOut(2, high=True)

app.add_components(rot, gruen)

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

app.run(message=on_message)
```

Beim Erzeugen der roten Led machen wir keine Angaben zum Startzustand. Default ist high=False, deshalb bleibt die Led dunkel. Bei der grünen Led geben wir high=True an, deshalb leuchtet die Led.

- **Die grüne Led soll jetzt mit 100 ms EIN und 900 ms AUS pulsieren.**
- **Die grüne Led soll mit der Frequenz von 1 Hz blinken.**

```
app.add_components(rot, gruen)

rot.pulse = 100, 900
gruen.blink = 1
```

- **Nach 5 Sekunden soll die rote Led dauernd eingeschaltet bleiben.**
- **Gleichzeitig soll die grüne Led ausgehen.**

```
gruen.blink = 1
app.after(5000, rot.set_high, True)
app.after(5000, gruen.set_high, False)
```

Da im ***after*** - Befehl nur der Funktionsname übergeben werden kann, ist das Property ***high*** hier nicht zu gebrauchen. Die Komponente DigitalOut stellt uns aber die Setter - Funktion ***set_high()*** zur Verfügung.

- **Nach total 7 Sekunden soll die rote Led ihren Zustand wechseln. Dazu verwenden wir die *toggle()* - Funktion.**

```
app.after(5000, gruen.set_high, False)
app.after(7000, rot.toggle)
```

Wie wir sehen, wird bei jedem Zustandswechsel eine 'changed' Nachricht erzeugt.

```
<DigitalOut object at 2000ee00> changed 1
<DigitalOut object at 2000ee00> changed 0
<DigitalOut object at 2000ed10> changed 0
...
```

## PWMOut (PWMOut.py)

- **Wir erstellen ein Programm mit einer roten Led am PWM - Ausgang 0.**
- **Die rote Led soll mit einer Frequenz von 1 Hz blinken.**
- **Alle Nachrichten werden auf der Konsole ausgegeben.**

```
from MMF_RP2040 import *

app = Application()

rot = PWMOut(0)

app.add_components(rot)

rot.blink = 1

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

app.run(message=on_message)
```

Anstelle von DigitalOut wird hier PWMOut verwendet. Auf den ersten Blick sieht alles ganz normal aus. Wenn man genauer hinschaut, fällt aber auf, dass die Led weniger stark leuchtet. Das liegt daran, dass ohne Angabe eines Wertes ein PWM - Verhältnis von 50% verwendet wird.

- **Wir schliessen wir ein Potentiometer an den analogen Eingang 0 an.**
- **Mit dem Potentiometer soll die Helligkeit gesteuert werden.**

Zuerst erzeugen wir einen analogen Eingang. Dieser wird später im Detail besprochen.

```
rot = PWMOut(0)
input_a = AnalogIn(0)

app.add_components(rot, input_a)
```

Jetzt benötigen wir eine Funktion, die den Potentiometerstand ausliest und damit den PWM - Wert setzt. Für beide Werte wird Prozent verwendet.

```
def set_brightness():
    rot.percent = input_a.percent
```

Diese Funktion soll alle 100 ms aufgerufen werden.

```
app.add_function(100, set_brightness)
```

- **Nach 10 Sekunden soll die rote Led ausgehen.**
- **Nach weiteren 2 Sekunden soll sie ohne blinken einschalten.**
- **Nochmals 2 Sekunden später wird das Programm beendet.**

```
app.add_function(100, set_brightness)
app.after(10000, rot.set_high, False)
app.after(12000, rot.set_high, True)
app.after(14000, app.stop)
```

# Digitale Eingänge

## DigitalIn und Button (DigitalIn.py)

Ein digitaler Eingang kann HIGH oder LOW sein. DigitalIn gibt 1 für HIGH und 0 für LOW zurück.
Ein Button ist an einen digitalen Eingang angeschlossen, gibt aber True zurück, wenn er gedrückt wird.

Wir verwenden diesmal ein einfaches Programm mit einem digitalen Input und einem Button.

```
from MMF_RP2040 import *

app = Application()

input_a = DigitalIn(12)
button_b = Button(13)

app.add_components(input_a, button_b)


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


print('input_a beim Start', input_a.value)
print('button_b beim Start', button_b.value)

app.run(message=on_message)
```

**DigitalIn**
Das Erstellen der Komponente ist sehr einfach:
`input_a = DigitalIn(12)`

Dabei wird der interne PullUp - Widerstand aktiviert. Falls das nicht gewünscht ist, muss die Komponente mit
`input_a = DigitalIn(12, pullup=False)`
erstellt werden.

Das Ergebnis kann mit ***comp.value*** abgefragt werden. Falls der Eingang HIGH ist, wird 1 zurückgegeben, andernfalls 0.

Da am Port 12 der Taster A angeschlossen ist und dieser in gedrücktem Zustand mit GND verbindet, wird im Ruhezustand 1 und bei gedrückten Taster 0 zurückgegeben.

Jede Änderung löste eine ***'changed' - Nachricht*** aus. In ***data*** steht der neue Zustand.

**Button**
Da an eine digitalen Eingang oft ein Taster angeschlossen ist, gibt es hierfür eine eigene Komponente.
`button_b = Button(13)`

Dabei wird der interne PullUp - Widerstand aktiviert. Ausserdem wird angenommen, dass der Button bei Betätigung eine Verbindung zu GND herstellt.

Wie bei DigitalIn kann der interne PullUp mit ***pullup=False*** ausgeschaltet werden. Der Benutzer muss dann aber durch die Hardware selbst sicherstellen, dass immer ein eindeutiger Zustand vorhanden ist. Wenn der interne PullUp ausgeschaltet wird, nimmt die Komponente an, dass der Druck auf den Taster einen HIGH - Zustand erzeugt.

Das kann auch explizit gesteuert werden.
`button_b = Button(13, reverse=False)`
nimmt in jedem Fall an, dass ein Tastendruck einen HIGH - Pegel anlegt.
Mit ***reverse=True*** wird angenommen, dass ein Tastendruck einen LOW - Pegel anlegt.

Das Ergebnis kann mit ***comp.value*** abgefragt werden. Falls der Taster gedrückt ist, wird True zurückgegeben, andernfalls False.

Jede Änderung löste eine ***'changed' - Nachricht*** aus. In ***data*** steht der neue Zustand.

Es werden zwei zusätzliche Nachrichten generiert.
***'pressed'*** wenn der Taster gedrückt wird,
***'released'*** wenn der Taster losgelassen wird.
Bei 'pressed' ist ***data*** irrelevant, bei 'released' wird in ***data*** die Dauer des Drucks in Millisekunden übermittelt.

## Analoge Eingänge

Analoge Eingänge können die Spannung am Eingang messen. Ohne zusätzliche Beschaltung am Eingang muss diese Spannung zwischen 0V und der Betriebsspannung liegen.

### AnalogIn (AnalogIn.py)

Diese Komponente ist controllerabhängig. Die Angaben hier beziehen sich auf den RP2040.
Wir haben die Komponente bereits im letzten Kapitel verwendet. Jetzt betrachten wir sie etwas genauer.

Ein analoger Eingang ist in der Lage eine Spannung zwischen 0V und der Eingangsspannung (hier 3.3V) zu messen.

```
from MMF_RP2040 import *

app = Application()

input_a = AnalogIn(0)

app.add_components(input_a)

def messung():
    value = input_a.value
    volt = input_a.volt
    percent = input_a.percent
    print(f'{value} entspricht {volt} Volt ({percent}% der Betriebsspannung.')

app.add_function(500, messung)

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

app.run(message=on_message)
```

Die Werte können direkt ermittelt werden.
***value*** ist ein controllerabhängiger Wert, beim RP2040 ist es eine ganze Zahl zwischen 0 und 65535.
***volt*** gibt die gemessene Spannung zurück.
***percent*** gibt einen Prozentwert zurück. 0% entspricht 0V, 100% der Betriebsspannung.