Inhaltsverzeichnis

Grundsätzliches

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

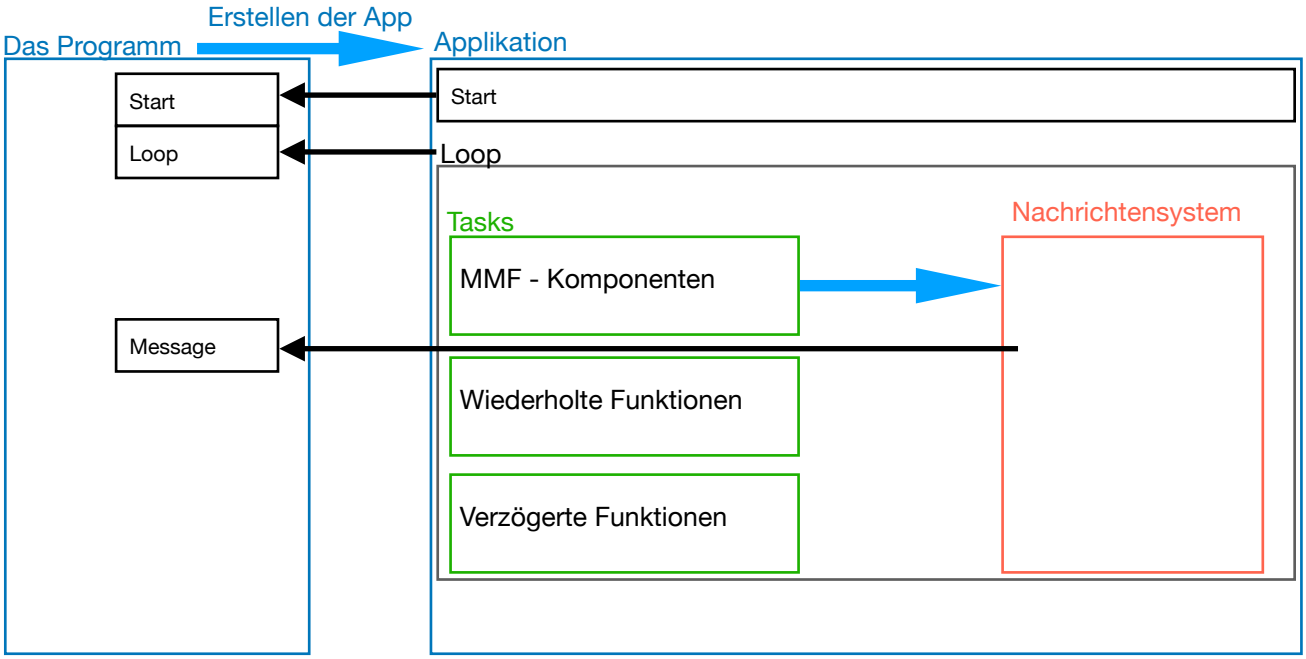
Programme in Python sind sehr schlecht für Echtzeitapplikationen geeignet. Oft ist das aber auch nicht notwendig. Es genügt, wenn gewisse Vorgänge scheinbar selbstständig im Hintergrund ablaufen.

Dieses Handbuch richtet sich an Einsteiger, die das Framework kennenlernen möchten. Es werden bewusst nicht alle Aspekte behandelt, um die Sache übersichtlich und verständlich zu halten. Es werden nur Standardkomponenten verwendet. Dieses Handbuch ist ein Referenzhandbuch und kein Tutorial. Pythonkenntnisse werden vorausgesetzt.
Das Erstellen eigener Komponenten wird im Programmierhandbuch für Fortgeschrittene behandelt.

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. Als Beispiel ist hier ein RP2040 (Raspberry Pi Pico) implementiert ***(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 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 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.

Die Basisklassen

Importiert wird immer das prozessorabhängige Modul. In unserem Beispiel ist das ***MMF_RP2040.py***. Wenn ein weiteres hardwareabhängiges Modul vorhanden ist (hier ***MMF_Explorer.py***) kann auch dieses importiert werden. Diese Importe stellen auch alle Funktionen der hardwareunabhängigen Module zur Verfügung.

Application

Diese Klasse erzeugt eine Applikation. Die Applikation verwaltet den Code und erlaubt die Ausführung scheinbar gleichzeitig ablaufender Vorgänge.

Es muss immer genau eine einzige Instanz der Applikationsklasse existieren. Diese wird ohne Argumente mit **`app = Application()`** erzeugt. Ein zweiter Versuch eine Instanz zu erstellen, führt zu einem Fehler.

Die Klasse kann zu jeder Zeit die vorher erstellte Instanz liefern.
`app = Application.app()`

Klassenmethoden

.app()	app = Application.app()	Gibt die früher erstellte Instanz zurück. Wenn noch keine Instanz vorhanden ist, wird None zurückgegeben.
.millis()	ms = Application.millis() ms = app.millis()	Gibt eine aufsteigende Anzahl Millisekunden zurück.

millis() kann sowohl über die Klasse (ms = Application.millis()), wie auch über eine Instanz (ms = app.millis()) aufgerufen werden.

Instanzmethoden

.run(message=None)	app.run() app.run(message=on_messsage)	Die Applikation wird gestartet. Das ist immer der letzte Befehl im Programm. Optional kann der Name einer Funktion für die Nachrichtenbehandlung angegeben werden.
.stop()	app.stop()	Das Programm wird beendet.
.clear()	app.clear()	Alle Tasks werden gelöscht.
.add_components(*args)	app.add_components(led1, motor1, ...)	Alle MMF - Komponenten müssen der Applikation hinzugefügt werden.
.add_function(interval, func, *args, **kwargs)	interval: Intervall in Millisekunden func: Funktionsname *args: Positionsargumente **kwargs: benannte Argumente	Hinzufügen einer Funktion, die in bestimmten Zeitabständen ausgeführt werden soll.
.after(time, func, *args, **kwargs)	time: Zeit in ms func: Funktionsname *args: Positionsargumente **kwargs: benannte Argumente	Es wird ein Auftrag an die Applikation erteilt, eine Funktion in ***time*** Millisekunden einmalig auszuführen.
.notify(sender, topic, data)	app.notify(button1, 'pressed', 1)	Eine Meldung an den Nachrichtenempfänger senden.

Settings
app.settings ist ein Dictionary, das beliebig bearbeitet und benutzt werden kann. Optional kann dieses Dictionary mit ***app.save_settings()*** permanent gespeichert werden. Das Dictionary steht dann beim nächsten Start wieder unverändert zur Verfügung.

Task
Tasks sind Aufgaben, die von der Applikation selbstständig ausgeführt werden. Dabei enthält der Task Instruktionen, die von der Applikation berücksichtigt werden. Es wird nie eine direkte Instanz von Task erstellt. Standardmässig sind drei Tasktypen vorhanden.

Verzögerte Funktion
Das ist der Auftrag an die Applikation, eine Funktion nach einer bestimmten Zeit selbstständig auszuführen. Der Task wird automatisch nach der Ausführung gelöscht. Diese Auftragserteilung erfolgt sofort, blockiert also das System nicht.

Ein Beispiel:

```
def info(a, text='Verzögerung in Sekunden: '):
    print(text + str(a))

app.after(2000, info, 2)
```

Ruft nach 2 Sekunden info(2) auf und gibt den Text
`Verzögerung in Sekunden: 2`
aus.

```
app.after(3000, info, 3, text='Sekunden')
```

Ruft nach 3 Sekunden info(3, text='Sekunden') auf und gibt den Text
`Sekunden: 3`
aus.

app.after gibt den erzeugten Task zurück. Das ist aber nicht relevant, da der Task nach Ausführung sowieso gelöscht wird.

Formal erfolgt der Aufruf von ***after*** wie folgt:

```
task = app.after(time, func, *args, **kwargs)
```

Der Rückgabewert ***task*** ist irrelevant, da er nur kurz lebt. Deshalb wird er üblicherweise nicht entgegengenommen.

time ist die Verzögerungszeit in Millisekunden.

func ist der Funktionsname ohne Klammern und Argumente.

****args*** sind die Positionsargumente der aufzurufenden Funktion.

*****kwargs*** sind die benannten Argumente der aufzurufenden Funktion.

Wiederholte Funktion
Das ist der Auftrag an die Applikation, eine Funktion in einem bestimmten Intervall immer wieder aufzurufen. Diese Auftragserteilung erfolgt sofort, blockiert also das System nicht.

Ein Beispiel:

```
app.zaehlerstand = 0

def zaehle(step):
    app.zaehlerstand += step

task = app.add_function(1000, zaehle, 2)
```

Der Zählerstand wird im Sekundentakt (1000 ms) um 2 hochgezählt.
Dieser Funktionsaufruf erfolgt dauernd, muss also manuell gestoppt werden. Das ist über den zurückgegebenen Task möglich. Mit ***task.stop()*** wird der Task entfernt.

Formal erfolgt der Aufruf von ***add_function*** wie folgt:

```
task = app.add_function(interval, func, *args, **kwargs)
```

Der Rückgabewert ***task*** kann zur Beeinflussung des Tasks verwendet werden.

interval ist das Aufrufintervall in Millisekunden.

func ist der Funktionsname ohne Klammern und Argumente.

****args*** sind die Positionsargumente der aufzurufenden Funktion.

*****kwargs*** sind die benannten Argumente der aufzurufenden Funktion.

Einflussmöglichkeiten auf Tasks
Alle Tasks, die mehr als einmal ausgeführt werden, können von aussen beeinflusst werden.

task.stop()	Stoppt den Task und entfernt ihn aus der Applikation.
task.active = False	Stoppt den Task, entfernt ihn aber nicht.
task.active = True	Ein gestoppter Task wird wieder aktiviert.
task.interval = 200	Ein neues Intervall wird gesetzt.

Die nachfolgend besprochenen Komponententasks besitzen weitere Einflussmöglichkeiten.
Es gibt diverse Standardkomponenten, die in jedem System vorhanden sind. Fortgeschrittene Benutzer haben die Möglichkeit, eigene Komponenten zu erstellen.

Die MMF - Komponenten

MMF - Komponenten sind Tasks, die üblicherweise Hardwarekomponenten ansprechen. In diesem Handbuch besprechen wir nur die Standardkomponenten, die in jedem System vorhanden sein sollten. Alle verwendeten Komponenten müssen mit

```
app.add_components(comp1, comp2, ..)
```

der App hinzugefügt werden.

DigitalOut

Das ist eine Grundfunktion jedes Microcontrollers. Ein digitaler Ausgang kann entweder der Zustand HIGH oder LOW haben.

Erzeugen der Komponente

```
comp = DigitalOut(num, high=False)
```

num gibt die Nummer des Ausgangs an, an den der digitale Wert ausgegeben werden soll.
high setzt den Ausgang beim Start. Bei high=True ist der Ausgang HIGH, sonst ist er LOW.

Eigenschaften und Befehle

comp.high = True comp.high = False	Ausgang auf HIGH oder LOW setzen. Die Benutzung dieser Befehle stoppt blink- und pulse- Aufträge.
comp.blink = 2	Blinken mit einer Frequenz von 2 Hz. Stoppt pulse.
comp.pulse = 500, 1500	Ausgabe von Pulsen mit 500 ms HIGH und 1500 ms LOW. Stoppt blink.
comp.toggle()	Zustand sofort wechseln. Stoppt blink und pulse.
comp.set_high(value) comp.set_blink(value) comp.set_pulse(value)	Funktionen zum Setzen der Properties.

Nachrichten

'changed'	Jeder Zustandswechsel erzeugt eine 'changed' Meldung. In data wird der neue Zustand übergeben.

DigitalIn

Das ist eine Grundfunktion jedes Microcontrollers. Ein digitaler Eingang kann entweder der Zustand 1 oder 0 haben.

Erzeugen der Komponente

```
comp = DigitalIn(num, pullup=True)
```

num gibt die Nummer des Ausgangs an, an den der digitale Wert ausgegeben werden soll.
pullup die meisten Mikrocontroller haben die Möglichkeit einen internen Pullup - Widerstand hinzuzuschalten. Dann ist der Zustand des offenen Eingangs immer 1.

Eigenschaften

comp.value	Zustand des Eingangs: 1 oder 0 Der Wert kann nur gelesen werden.

Nachrichten

'changed'	Jeder Zustandswechsel erzeugt eine 'changed' Meldung. In data wird der neue Zustand übergeben.

Button

Das ist ein Spezialfall einer DigitalIn - Komponente. Es wird angenommen, dass der Button in unbetätigtem Zustand den Eingang offen lässt und ihn bei Betätigung auf einen definierten Zustand zieht.

Erzeugen der Komponente

```
comp = Button(num, pullup=True)
```

num gibt die Nummer des Ausgangs an, an den der digitale Wert ausgegeben werden soll.
pullup die meisten Mikrocontroller haben die Möglichkeit einen internen Pullup - Widerstand hinzuzuschalten. In diesem Fall muss der Taster bei Betätigung eine Verbindung zu GND herstellen.

Eigenschaften

comp.value	True zeigt einen gedrückten Taster an. Der Wert kann nur gelesen werden.

Nachrichten

'changed'	Jeder Zustandswechsel erzeugt eine 'changed' Meldung. In data wird der neue Zustand übergeben.
'pressed'	Der Taster wurde gedrückt.
'released'	Der Taster wurde losgelassen. In data wird die Dauer des Drucks in Millisekunden übergeben.

AnalogIn

Diese Komponente ist controllerabhängig. Die Angaben hier beziehen sich auf den RP2040.

Erzeugen der Komponente

```
comp = AnalogIn(num)
```

num gibt die Nummer des Analogeingangs an.

Eigenschaften

comp.value	Das ist der rohe eingelesene 16 - Bit wert. Der Wert kann nur gelesen werden.
comp.volt	Das ist die Spannung in Volt am Eingang. Ihr maximaler Wert ist die Betriebsspannung (3.3 V). Der Wert kann nur gelesen werden.
comp.percent	Die Spannung am Eingang in Prozent. 0 V entspricht 0%, die Betriebsspannung (hier 3.3 V) entspricht 100%. Der Wert kann nur gelesen werden.

Nachrichten

keine

PWMOut

Diese Komponente ist controllerabhängig. Die Angaben hier beziehen sich auf den RP2040.
Diese Komponente ist von ***DigitalOut*** abgeleitet und unterstützt dessen Eigenschaften und Befehle.

Erzeugen der Komponente

```
comp = PWMOut(num, percent=50, freq=500)
```

num gibt die Nummer des Ausgangs an.
percent ist der Anteil in %, an dem das Signal HIGH ist.
freq in Hz bestimmt die Periodendauer.

Es wird ein Rechtecksignal mit der Frequenz ***freq*** ausgegeben. ***percent*** bestimmt das Verhältnis zwischen HIGH und LOW.

Eigenschaften und Befehle

comp.percent	Prozentsatz (float) des HIGH-Zustands.
comp.freq	Die Frequenz in Hz bestimmt die Periodendauer. Die Standardfrequenz von 500 Hz ist ideal zur Ansteuerung von Servos.
comp.high = True comp.high = False	Ausgang auf HIGH oder LOW setzen. Die Benutzung dieser Befehle stoppt blink- und pulse- Aufträge.
comp.blink = 2	Blinken mit einer Frequenz von 2 Hz. Stoppt pulse.
comp.pulse = 500, 1500	Ausgabe von Pulsen mit 500 ms HIGH und 1500 ms LOW. Stoppt blink.
comp.toggle()	Zustand sofort wechseln. Stoppt blink und pulse.
comp.set_high(value) comp.set_blink(value) comp.set_pulse(value)	Funktionen zum Setzen der Properties.

Nachrichten

alle Nachrichten von DigitalOut

Anhang

Tutorials

Für den Einstieg wird das Einführungstutorial (***MMFEinführungstutorial***) empfohlen. Es führt in den Umgang mit dem Framework ein und zeigt an einigen Beispielen, wie das Multitasking funktioniert.

Das Komponententutorial (***MMFKomponententutorial***) ist der nächste Schritt. Es zeigt in der Praxis, wie die hier beschriebenen Komponenten angewendet werden.

Lizenz