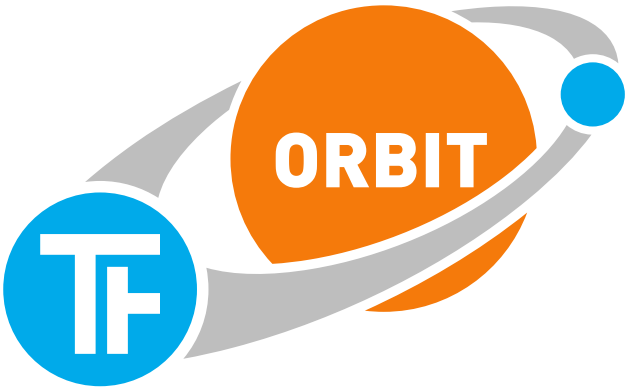

orbit Documentation

Release 0.1.0 alpha

Tobias Kiertscher

26.06.2014

Inhaltsverzeichnis

KAPITEL 1

Architektur einer ORBIT-Anwendung

Eine ORBIT-Anwendung besteht aus einem Kern und assoziierten Jobs. Jobs können Dienste oder Apps sein. Die Dienste werden automatisch beim Start des Kerns aktiviert und sind gleichzeitig aktiv. Von den Apps wird nur die Standard-App beim Start des Kerns akiviert und zu jedem Zeitpunkt kann immer nur eine App aktiv sein.

1.1 Jobs und Komponenten

Jobs werden aus Komponenten zusammengesetzt. Alle anwendungsspezifischen Fähigkeiten können in eigenen Komponenten und Jobs implementiert werden. Für einige Standardaufgaben bringt ORBIT *Integrierte Komponenten* und *Integrierte Jobs* mit.

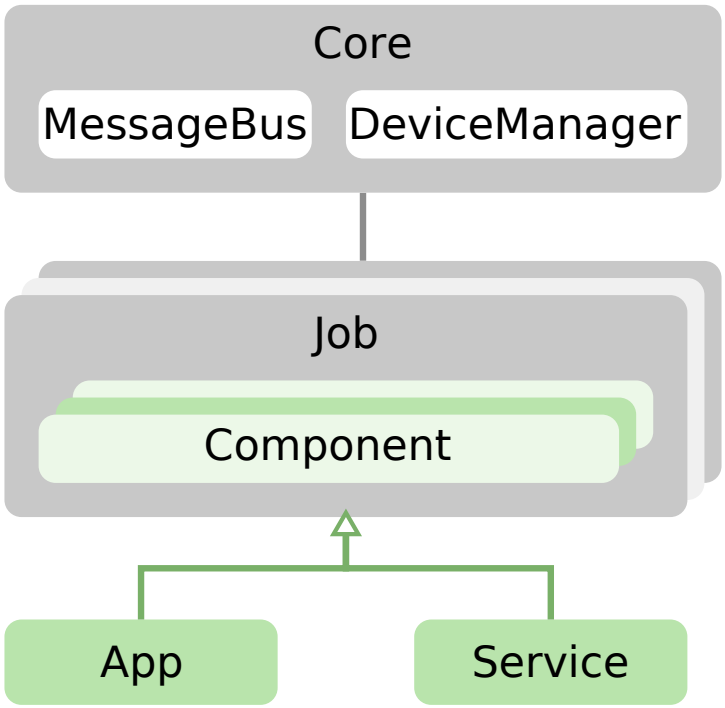

Abbildung 1.1: Eine grobe Übersicht über die Architektur einer ORBIT-Anwendung

Siehe auch:
`orbit_framework.Core`,
`orbit_framework.Component`,

orbit_framework.App, orbit_framework.Service

1.2 Gerätemanager

Der Gerätemanager im Kern verwaltet die TinkerForge-Verbindungen und die angeschlossenen Brick(let)s. Jede Komponente kann die Zuordnung von ein oder mehreren Brick(let)s eines Typs anfordern. Dabei können auch UIDs von Brick(let)s als Einschränkung angegeben werden. Sobald ein Brick(let) verfügbar ist, wird es den entsprechenden Komponenten zugeordnet und die Komponenten werden über die Verfügbarkeit benachrichtigt. Wird eine Verbindung getrennt, wird den Komponenten das Brick(let) wieder entzogen.

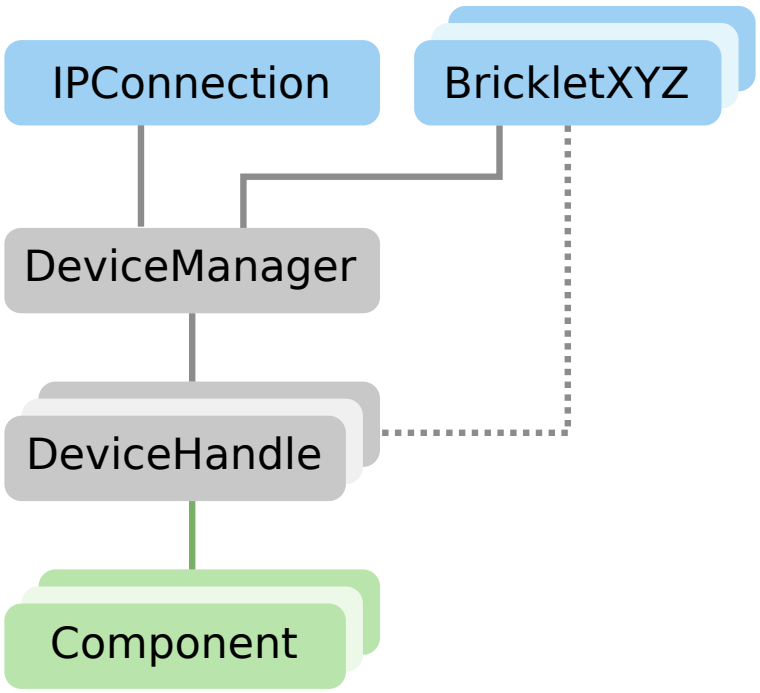

Abbildung 1.2: Eine Übersicht über den Gerätemanager

Siehe auch:
orbit_framework.Component.add_device_handle,
orbit_framework.devices.DeviceHandle,
orbit_framework.devices.DeviceManager

1.3 Nachrichtensystem

Komponenten und Jobs können über ein integriertes, asynchrones Nachrichtensystem kommunizieren. Das Nachrichtensystem ermöglicht eine weitgehende Entkopplung der Komponenten und Jobs von einander und sorgt für eine robuste Anwendung.

Siehe auch:
orbit_framework.Component.add_listener(), orbit_framework.Job.add_listener(),
orbit_framework.Component.send(), orbit_framework.Job.send(),

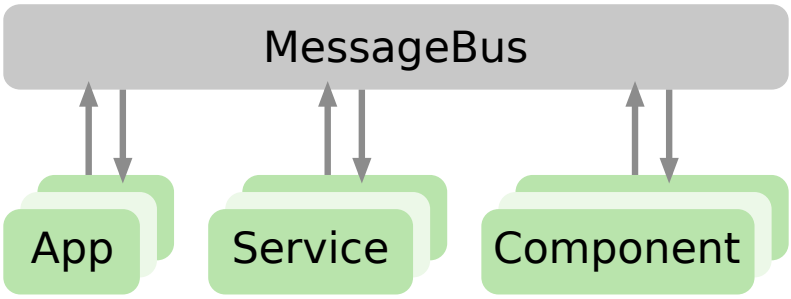

Abbildung 1.3: Eine Übersicht über das Nachrichtensystem

`orbit_framework.messaging.MessageBus`

1.4 Übersicht

Die folgende Übersicht stellt die wesentlichen Objekte in einer ORBIT-Anwendung und deren Assoziationen dar. Grau hinterlegte Objekte werden von ORBIT implementiert. Blau hinterlegte Objekte werden durch die TinkerForge-Python-Bibliothek implementiert. Grün hinterlegte Objekte werden anwendungsspezifisch implementiert, wobei Basisklassen die die Implementierung erleichtern.

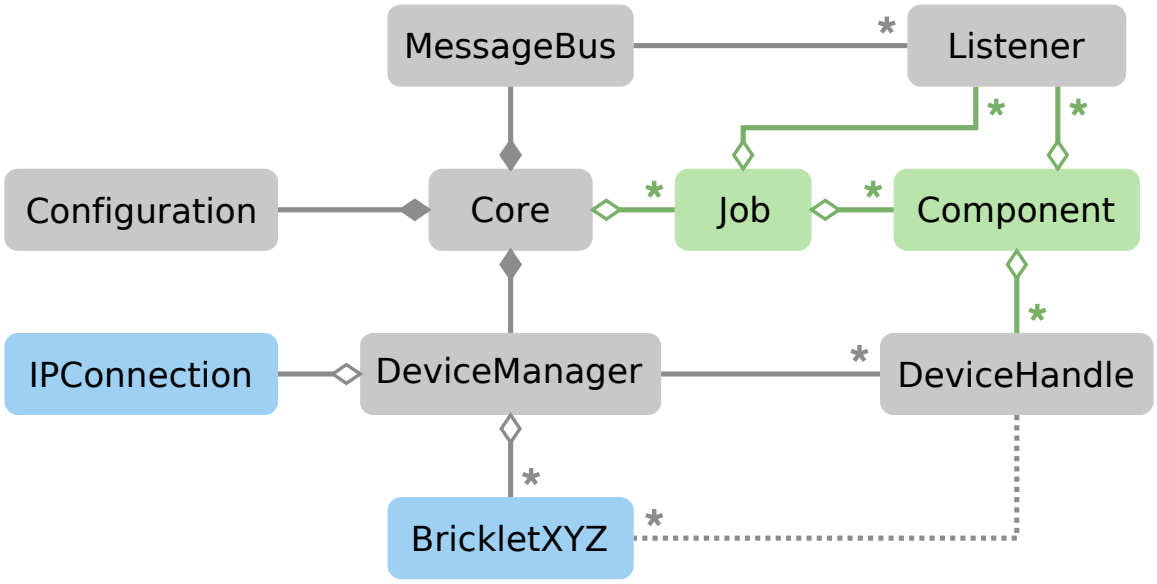

Abbildung 1.4: Eine detaillierte Übersicht über die Architektur einer ORBIT-Anwendung

KAPITEL 2

Setup

2.1 Voraussetzungen

2.1.1 Software

ORBIT unterstützt Python 2.6+ und Python 3.x.

Für die Installation werden die setuptools[1] benötigt, die den Installationsbefehl *easy_install* zur Verfügung stellen, mit dem Python-Eggs installiert werden können.

ORBIT setzt die TinkerForge-Python-Bindings voraus. Diese werden auf der Website von TinkerForge als Download zur Verfügung gestellt. Das im Download enthaltene Python-Egg `tinkerforge.egg` muss mit *easy_install* installiert werden.

- Dokumentation der Python-Bindings[2]
- Download-Archiv[3]

2.1.2 Hardware

Für die Verwendung von TinkerForge ist mindestens ein Master-Brick und ein Bricklet erforderlich. Wenn der Master-Brick per USB an den Computer (PC, Raspberry PI, o.a.) angeschlossen werden soll, muss der Brick-Deamon[4] installiert sein.

Einige der Beispiele verwenden ein LCD-20x4-Display-Bricklet und ein Motion-Detector-Bricklet.

2.2 Installation

ORBIT wird, wie die TinkerForge-Bindings, als Python-Egg installiert. Die Datei `orbit_framework.egg` kann als Release[5] heruntergeladen werden.

Anschließend kann ORBIT mit *easy_install* installiert werden. Dazu wird *easy_install* wie folgt aufgerufen:

```
> easy_install orbit_framework.egg
```

Unter Umständen muss der Befehl mit Administrator- bzw. Root-Rechten (z.B. mit `sudo`) aufgerufen werden.

Nun ist ORBIT installiert und es können die *Beispiele* ausgeführt werden.

[1] https://pypi.python.org/pypi/setuptools
[2] http://www.tinkerforge.com/de/doc/Software/API_Bindings_Python.html
[3] http://download.tinkerforge.com/bindings/python/
[4] http://www.tinkerforge.com/de/doc/Software/Brickd.html
[5] https://github.com/mastersign/orbit/releases

KAPITEL 3

Beispiele

Die Python-Quelltexte der Beispiele befinden sich im Verzeichnis `examples`. Für die Ausführung der Beispiele muss ORBIT als Python-Package installiert sein (Siehe *Setup*), Und es muss ein Brick-Deamon laufen. Mit der Standardkonfiguration versucht ORBIT den Brick-Deamon unter `localhost:4223` zu erreichen.

Einige Beispiele setzen ein LCD-20x4-Bricklet und einige auch ein Motion-Detector-Bricklet voraus. Die Hardware-Voraussetzungen sind für jedes Beispiel extra angegeben.

3.1 Grundlagen

3.1.1 1. Basic

Quellcode Beispiel 1[1]

Hardware: keine

Beschreibung: Grundaufbau einer ORBIT-Anwendung mit einem Dienst.

Das Beispiel demonstriert die Einrichtung, das Starten und Stoppen des Anwendungskerns. Darüber hinaus zeigt es, wie mit den beiden integrierten Komponenten `orbit_framework.components.common.EventCallbackComponent` und `orbit_framework.components.timer.IntervalTimerComponent` in regelmäßigen Abständen eine Funktion aufgerufen werden kann. Die beiden Komponenten werden in einem Dienst zusammengefasst, damit sie zur gesamten Laufzeit der Anwendung aktiv sind. Die Timer-Komponente sendet in regelmäßigen Abständen eine Nachricht über das ORBIT-Nachrichtensystem. Die Callback-Komponente wird so eingerichtet, dass sie die Timer-Nachrichten empfängt und beim Eintreffen einer Nachricht die Funktion aufruft.

3.1.2 2. Apps

Quellcode Beispiel 2[2]

Hardware: LCD 20x4 Display

Beschreibung: Eine ORBIT-Anwendung mit einem Dienst und zwei integrierten Apps.

Das Beispiel demonstriert, wie ein Dienst Tasten-Ereignisse von den LCD-Tasten über das Nachrichtensystem versendet und zwei Apps abhängig von diesen Nachrichten aktiviert werden. Dabei kommt in dem Dienst die integrierte Komponente `orbit_framework.components.lcd.LCD20x4ButtonsComponent` zum Einsatz. Für die zwei Apps wird die integrierte App `orbit_framework.jobs.apps.MessageApp` verwendet. Darüber hinaus wird gezeigt, wie eine App als Standard-App für die ORBIT-Anwendung eingerichtet wird.

[1] https://github.com/mastersign/orbit/blob/master/examples/001_basic.py
[2] https://github.com/mastersign/orbit/blob/master/examples/002_apps.py

3.1.3 3. Component

Quellcode Beispiel 3[3]

Hardware: LCD 20x4 Display

Beschreibung: Eine ORBIT-Anwendung mit einer eigenenen Komponente.

Das Beispiel demonstriert, wie eine eigene Komponente für die Interaktion mit einem TinkerForge-Bricklet implementiert und verwendet wird. Die Komponente fängt die Tasten-Ereignisse eines LCD-Displays ab, zählt dabei einen Zähler hoch und gibt den aktuellen Zählerstand auf dem Display aus. Zusätzlich wird gezeigt, wie eine eigene App-Klasse implementiert wird.

[3] https://github.com/mastersign/orbit/blob/master/examples/003_component.py

KAPITEL 4

API-Referenz

Die öffentliche ORBIT-API besteht aus den folgenden Modulen:

4.1 Modul orbit_framework

Diese Modul enthält die wichtigsten Klassen für die Entwicklung einer ORBIT-Anwendung. Eine ORBIT-Anwendung wird von einem `Core` verwaltet und kann mehrere `Job`-Objekte enthalten. Jeder `Job` fasst eine Gruppe von `Component`-Objekten zusammen. Ein `Job` ist entweder eine `App` oder ein `Service`.

Die TinkerForge-Bricklets werden durch den `devices.DeviceManager` verwaltet und den `Component`-Objekten über `devices.DeviceHandle`-Objekte zugeordnet.

`Component`- und `Job`-Objekte kommunizieren asynchron über das ORBIT-Nachrichtensystem welches durch die Klasse `messaging.MessageBus` implementiert wird.

`Component`- und `Job`-Objekte können jederzeit Nachrichten senden. Diese Nachrichten werden in einer Warteschlange abgelegt und in einem dedizierten Nachrichten-Thread an die Empfänger versandt (`Job.send()`, `Component.send()`).

Für den Empfang von Nachrichten werden `messaging.Listener`-Objekte verwendet. Ein `messaging.Listener` bindet ein Empfangsmuster/-filter an ein Callback (`messaging.Slot`, `Job.add_listener()`, `Component.add_listener()`). Wird eine Nachricht über das Nachrichtensystem versandt, welches dem Empfangsmuster entspricht, wird das Callback des Empfängers aufgerufen.

Das Modul enthält die folgenden Klassen:

- `Core`
- `Job`
- `App`
- `Service`
- `Component`

4.1.1 Core

class `orbit_framework.`**`Core`**(*config = Configuration()*)

Diese Klasse repräsentiert den Kern einer ORBIT-Anwendung.

Parameter

`config` **(*optional*)** Die Konfiguration für die ORBIT-Anwendung. Eine Instanz der Klasse `setup.Configuration`.

Beschreibung

Diese Klasse ist verantwortlich für die Einrichtung des Nachrichtensystems und der Geräteverwaltung. Des Weiteren verwaltet sie alle Dienste und Apps, die in der ORBIT-Anwendung verwendet werden.

Für die Einrichtung einer Anwendung wird zunächst eine Instanz von `Core` erzeugt. Dabei kann optional eine benutzerdefinierte Konfiguration (`setup.Configuration`) an den Konstruktor übergeben werden. Anschließend werden durch den mehrfachen Aufruf von `install()` Jobs hinzugefügt. Jobs können Dienste (`Service`) oder Apps (`App`) sein. Über die Eigenschaft `default_application` kann eine App als Standard-App festgelegt werden.

Um die ORBIT-Anwendung zu starten wird die Methode `start()` aufgerufen. Um die ORBIT-Anwendung zu beenden, kann entweder direkt die Methode `stop()` aufgerufen werden, oder es werden vor dem Start der Anwendung ein oder mehrere Slots (Ereignisempfänger für das ORBIT-Nachrichtensystem) hinzugefügt, welche die Anwendung stoppen sollen.

activate(*application*)
Aktiviert eine App. Die App kann direkt übergeben oder durch ihren Namen identifiziert werden.

Zu jedem Zeitpunkt ist immer nur eine App aktiv. Ist bereits eine andere App aktiv, wird diese deaktiviert, bevor die übergebene App aktiviert wird.

Ist die Eigenschaft `App.in_history` der bereits aktiven App gleich `True`, wird die App vor dem Deaktivieren in der App-History vermerkt.

Wird der Name einer App übergeben, die nicht in der ORBIT-Anwendung installiert ist, wird eine `KeyError` ausgelöst.

Siehe auch: `deactivate()`, `clear_application_history()`, `for_each_active_job()`, `Job.active`

add_stopper(*slot*)
Fügt einen `messaging.Slot` hinzu, der das Stoppen der ORBIT-Anwendung veranlassen soll.

Siehe auch: `remove_stopper()`

clear_application_history()
Leert die App-History.

Das hat zur Folge, dass nach dem Deaktivieren der zur Zeit aktiven App die Standard-App oder gar keine aktiviert wird.

Siehe auch: `activate()`, `deactivate()`

configuration
Gibt die ORBIT-Anwendungskonfiguration zurück. (*schreibgeschützt*)

Siehe auch: `setup.Configuration`

deactivate(*application*)
Deaktiviert eine App. Die App App kann direkt übergeben oder durch ihren Namen identifiziert werden.

Nach dem Deaktivieren der App wird geprüft, ob in der App-History eine App vermerkt ist, welche vorher aktiv war. Ist dies der Fall wird jene App automatisch aktiviert.

Ist die App-History leer, wird geprüft ob eine Standard-App registriert ist (`default_application`) und ggf. diese aktiviert.

Siehe auch: `activate()`, `clear_application_history()`, `Job.active`

default_application
Gibt die Standard-App zurück oder legt sie fest.

Siehe auch: `App`

device_manager
Gibt den Gerätemanager zurück. (*schreibgeschützt*)

Siehe auch: `devices.DeviceManager`

for_each_active_job(*f*)
Führt eine Funktion für jeden aktiven Job aus.

Siehe auch: `activate()`, `deactivate()`

for_each_job(*f*)
Führt eine Funktion für jeden installierten Job aus.

Siehe auch: `jobs`, `install()`, `uninstall()`

install(*job*)
Fügt der ORBIT-Anwendung einen Job (`Job`) hinzu. Ein Job kann ein Dienst (`Service`) oder eine App (`App`) sein. Jobs können vor oder nach dem Starten der ORBIT-Anwendung hinzugefügt werden.

Wird ein Job mehrfach hinzugefügt, wird eine Ausnahme vom Typ `AttributeError` ausgelöst.

Siehe auch: `uninstall()`, `jobs`, `App`, `Service`

is_started
Gibt an ob die ORBIT-Anwendung gestartet wurde oder nicht. (*schreibgeschützt*) Ist `True` wenn die Anwendung läuft, sonst `False`.

Siehe auch: `start()`, `stop()`

jobs
Gibt ein Dictionary mit allen installierten Jobs zurück. (*schreibgeschützt*) Der Schlüssel ist der Name des Jobs und der Wert ist der Job selbst.

Siehe auch: `Job`, `install()`, `uninstall()`

message_bus
Gibt das Nachrichtensystem zurück. (*schreibgeschützt*)

Siehe auch: `messaging.MessageBus`

remove_stopper(*slot*)
Entfernt einen `messaging.Slot`, der das Stoppen der ORBIT-Anwendung veranlassen sollte.

Siehe auch: `add_stopper()`

start()
Startet die ORBIT-Anwendung.

Beim Start wird das Nachrichtensystem gestartet und damit die Weiterleitung von Ereignissen zwischen den Jobs und Komponenten aktiviert. Des Weiteren wird der Gerätemanager gestartet, der die Verbindung zum TinkerForge-Server aufbaut und die angeschlossenen Bricks und Bricklets ermittelt.

Ist die Infrastruktur des Kerns erfolgreich gestartet, werden zunächst alle Dienste, und zum Schluss die Standard-App aktiviert.

Bemerkung: Wird diese Methode aufgerufen, wenn die ORBIT-Anwendung bereits gestartet wurde, wird lediglich eine Meldung auf der Konsole ausgegeben.

Siehe auch: `stop()`, `is_started`

stop()
Stoppt die ORBIT-Anwendung.

Beim Stoppen werden zunächst alle Jobs (Dienste und Apps) deaktiviert. Anschließend wird der Gerätemanager beendet und dabei die Verbindung zum TinkerForge-Server getrennt. Zum Schluss wird das Nachrichtensystem beendet und die Weiterleitung von Ereignissen gestoppt.

Bemerkung: Wird diese Methode aufgerufen, wenn die ORBIT-Anwendung nicht gestartet ist, wird lediglich eine Meldung auf der Konsole ausgegeben.

Siehe auch: `start()`, `is_started`

trace(*text*)
Schreibt eine Nachverfolgungsmeldung mit dem Ursprung `Core` auf die Konsole.

uninstall(*job*)
Entfernt einen Job aus der ORBIT-Anwendung. Ist der Job zur Zeit aktiv, wird er deaktiviert bevor er aus der Anwendung entfernt wird.

Wird ein Job übergeben, der nicht installiert ist, wird eine Ausnahme vom Typ `AttributeError` ausgelöst.

Siehe auch: `install()`, `jobs`

wait_for_stop()
Blockiert solange den Aufrufer, bis die ORBIT-Anwendung beendet wurde.

Der Grund für das Beenden kann ein direkter Aufruf von `stop()` oder das Abfangen eines Ereignisses von einem der Stopper-Slots sein.

Zusätzlich wird das Unterbrechungssignal (SIGINT z.B. Strg+C) abgefangen. Tritt das Unterbrechungssignal auf, wird die ORBIT-Anwendung durch den Aufruf von `stop()` gestoppt und die Methode kehrt zum Aufrufer zurück.

Bemerkung: Die Methode kehrt erst dann zum Aufrufer zurück,

- wenn alle Jobs deaktiviert wurden,
- der Gerätemanager alle Bricks und Bricklets freigegeben und die Verbindung zum TinkerForge-Server beendet hat
- und das Nachrichtensystem alle ausstehenden Ereignisse weitergeleitet hat und beendet wurde.

Warnung: Diese Methode darf nicht direkt oder indirekt durch einen `messaging.Listener` aufgerufen werden, da sie andernfalls das Nachrichtensystem der ORBIT-Anwendung blockiert.

Siehe auch: `stop()`, `add_stopper()`, `remove_stopper()`

4.1.2 Job

class `orbit_framework.`**Job**(*name*, *background*)
Dies ist die Basisklasse für Aufgaben in einer ORBIT-Anwendung.

Parameter

name Der Name der Aufgabe. Der Name muss innerhalb der ORBIT-Anwendung eindeutig sein.

background `True` wenn die Aufgabe als Hintergrunddienst laufen soll, sonst `False`‘.

Beschreibung

Ein Job wird durch die Zusammenstellung von Komponenten implementiert. Wird ein Job aktiviert, werden alle seine Komponenten aktiviert. Wird ein Job deaktiviert, werden alle seine Komponenten deaktiviert. Ein aktiver Job kann Nachrichten über das Nachrichtensystem austauschen. Ein inaktiver Job kann keine Nachrichten senden oder empfangen.

Bemerkung: Die `Job`-Klasse sollte nicht direkt verwendet werden. Statt dessen sollten die abgeleiteten Klassen `Service` und `App` verwendet werden.

Siehe auch: `add_component()`, `remove_component()`

active
Gibt einen Wert zurück oder legt ihn fest, der angibt, ob der Job aktiv ist.

Bemerkung: Dieses Attribut sollte nicht direkt gesetzt werden. Statt dessen sollte `Core.activate()` und `Core.deactivate()` verwendet werden.

Siehe auch: `Core.activate()`, `Core.deactivate()`, `on_activated()`, `on_deactivated()`

add_component(*component*)
Fügt dem Job eine Komponente hinzu.

Siehe auch: `remove_component()`, `components`, `Component`

add_listener(*listener*)
Richtet einen Nachrichtenempfänger für das Nachrichtensystem ein.

Als Empfänger wird üblicherweise ein `messaging.Listener`-Objekt übergeben.

Siehe auch: `messaging.Listener`, `messaging.MessageBus.add_listener()`, `remove_listener()`, `send()`

background
Legt fest, ob der Job als Hintergrunddienst ausgeführt wird oder als Vordergrund-App.

Mögliche Werte sind `True` für Hintergrunddienst oder `False` für Vordergrund-App.

components
Gibt ein Dictionary mit allen Komponenten des Jobs zurück. Die Namen der Komponenten werden als Schlüssel verwendet. Die Komponenten selbst sind die Werte.

Siehe auch: `add_component()`, `remove_component()`, `Component`

configuration
Gibt die Anwendungskonfiguration zurück. Wenn der Job nicht installiert ist, wird `None` zurück gegeben.

core
Gibt eine Referenz auf den Anwendungskern zurück. Wenn der Job nicht installiert ist, wird `None` zurück gegeben.

Siehe auch: `Core`, `Core.install()`

event_tracing
Legt fest, ob über das Nachrichtensystem versendete Nachrichten auf der Konsole protokolliert werden sollen.

Mögliche Werte sind `True` oder `False`.

Siehe auch: `send()`

for_each_component(*f*)
Führt die übergebene Funktion für jede Komponente des Jobs aus.

Siehe auch: `components`

name
Gibt den Namen des Jobs zurück.

on_activated()
Wird aufgerufen, wenn der Job aktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `active`, `Core.activate()`

on_core_started()
Wird aufgerufen, wenn der Anwendungskern gestartet wurde.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.start()`

on_core_stopped()
Wird aufgerufen, wenn der Anwendungskern gestoppt wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.stop()`

on_deactivated()
Wird aufgerufen, wenn der Job deaktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `active`, `Core.deactivate()`

on_install(*core*)
Wird aufgerufen, wenn der Job installiert wird.

Parameter

core Eine Referenz auf den Anwendungskern.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.install()`

on_uninstall()
Wird aufgerufen, wenn der Job deinstalliert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.uninstall()`

remove_component(*component*)
Entfernt eine Komponente aus dem Job.

Siehe auch: `add_component()`, `components`

remove_listener(*listener*)
Meldet einen Nachrichtenempfänger vom Nachrichtensystem ab.

Bemerkung: Es muss das selbe Empfängerobjekt übergeben werden, wie an `add_listener()` übergeben wurde.

Siehe auch: `messaging.MessageBus.remove_listener()`, `add_listener()`, `send()`

send(*name*, *value=None*)
Versendet eine Nachricht über das Nachrichtensystem.

Parameter

name Der Ereignisname für die Nachricht.

value (*optional*) Der Inhalt der Nachricht. Das kann ein beliebiges Objekt sein.

Beschreibung

Die Methode übergibt die Nachricht an das Nachrichtensystem und kehrt sofort zum Aufrufer zurück. Die Nachricht wird asynchron an die Empfänger übermittelt. Als Absender-Job wird der Name dieses Jobs eingetragen. Als Absenderkomponente wird `JOB` eingetragen.

Siehe auch: `add_listener()`, `remove_listener()`, `messaging.MessageBus.send()`

trace(*text*)
Schreibt eine Nachverfolgungsmeldung mit dem Ursprung `Service <Name>` oder `App <Name>` auf die Konsole.

tracing
Legt fest, ob Nachverfolgungsmeldungen für diesen Job auf der Konsole ausgegeben werden sollen.

Mögliche Werte sind `True` oder `False`.

Siehe auch: `trace()`

4.1.3 Service

class `orbit_framework.`**Service**(*name*)
Diese Klasse implementiert eine Hintergrundaufgabe in einer ORBIT-Anwendung.

Parameter

name Der Name der Aufgabe. Der Name muss innerhalb der ORBIT-Anwendung eindeutig sein.

Beschreibung

Hintergrundaufgaben werden üblicherweise direkt mit dem Starten der ORBIT-Anwendung aktiviert und erst mit dem Beenden der Anwendung wieder deaktiviert.

Diese Klasse erbt von `Job` und implementiert damit eine Aufgabe durch die Kombination von verschiedenen Komponenten. Diese Klasse kann direkt instanziert werden oder es kann eine Klasse abgeleitet werden.

Siehe auch: `Job`, `Job.add_component()`, `Core.install()`

4.1.4 App

class `orbit_framework.`**App**(*name*, *in_history=False*, *activator=None*, *deactivator=None*)
Diese Klasse implementiert eine Vordergrundaufgabe in einer ORBIT-Anwendung.

Parameter

name Der Name der Aufgabe. Der Name muss innerhalb der ORBIT-Anwendung eindeutig sein.

in_history (***optional***) Gibt an, ob diese App in der App-History berücksichtigt werden soll. (*Siehe auch:* `Core.activate()`, `Core.deactivate()`) Mögliche Werte sind `True` wenn die App in der App-History vermerkt werden soll, sonst `False`.

activator (***optional***) Slots für die Aktivierung der App. Ein einzelner `messaging.Slot`, eine Sequenz von Slot-Objekten oder `None`. (*Siehe auch:* `add_activator()`)

deactivator (***optional***) Slots für die Deaktivierung der App. Ein einzelner `messaging.Slot`, eine Sequenz von Slot-Objekten oder `None`. (*Siehe auch:* `add_deactivator()`)

Beschreibung

Diese Klasse erbt von `Job` und implementiert damit eine Aufgabe durch die Kombination von verschiedenen Komponenten. Diese Klasse kann direkt instanziert werden oder es kann eine Klasse abgeleitet werden.

Siehe auch: `Job`, `Job.add_component()`, `Core.install()`

activate()
Aktiviert die App.

Siehe auch: `Core.activate()`

add_activator(*slot*)
Fügt einen `messaging.Slot` für die Aktivierung der App hinzu.

Sobald eine Nachricht über das Nachrichtensystem gesendet wird, welche dem Empfangsmuster des übergebenen Slots entspricht, wird die App aktiviert.

Siehe auch: `remove_activator()`, `add_deactivator()`

add_deactivator(*slot*)
Fügt einen `messaging.Slot` für die Deaktivierung der App hinzu.

Sobald eine Nachricht über das Nachrichtensystem gesendet wird, welche dem Empfangsmuster des übergebenen Slots entspricht, wird die App deaktiviert.

Siehe auch: `remove_deactivator()`, `add_activator()`

deactivate()
Deaktiviert die App.

Siehe auch: `Core.deactivate()`

in_history
Gibt einen Wert zurück, der angibt, ob diese App in der App-History vermerkt wird oder nicht.

Mögliche Werte sind `True` oder `False`.

Siehe auch: `Core.activate()`, `Core.deactivate()`

on_install(*core*)
Wird aufgerufen, wenn die App installiert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.install()`, `Job.on_install()`

on_uninstall()
Wird aufgerufen, wenn die App deinstalliert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Core.uninstall()`, `Job.on_uninstall()`

remove_activator(*slot*)
Entfernt einen `messaging.Slot` für die Aktivierung der App.

Bemerkung: Es muss die selbe Referenz übergeben werden, wie an `add_activator()` übergeben wurde.

Siehe auch: `add_activator()`

remove_deactivator(*slot*)
Entfernt einen `messaging.Slot` für die Deaktivierung der App.

Bemerkung: Es muss die selbe Referenz übergeben werden, wie an `add_deactivator()` übergeben wurde.

Siehe auch: `add_deactivator()`

4.1.5 Component

class orbit_framework.**Component**(*name*)
Diese Klasse implementiert eine Komponente als Baustein für eine Aufgabe.

Parameter

name Der Name der Komponente. Der Name muss innerhalb der Aufgabe eindeutig sein.

Beschreibung

Komponenten sind der typische Weg, in einer ORBIT-Anwendung, mit einem TinkerForge-Brick(let) zu kommunizieren. Eine Komponente wird implementiert, indem eine Klasse von `Component` abgeleitet wird und im Konstruktur durch den Aufruf von `add_device_handle()` Geräteanforderungen eingerichtet werden.

Komponenten können über das Nachrichtensystem kommunizieren. Dazu können mit dem Aufruf von `add_listener()` Empfänger eingerichtet und mit dem Aufruf von `send()` Nachrichten versandt werden.

Siehe auch: `Job.add_component()`, `add_device_handle()`, `add_listener()`, `send()`

add_device_handle(*device_handle*)
Richtet eine Geräteanforderung ein.

Als Parameter wird ein `devices.SingleDeviceHandle` oder ein `devices.MultiDeviceHandle` übergeben.

Siehe auch: `remove_device_handle()`

add_listener(*listener*)
Richtet einen Nachrichtenempfänger für das Nachrichtensystem ein.

Als Empfänger wird üblicherweise ein `messaging.Listener`-Objekt übergeben.

Siehe auch: `messaging.Listener`, `messaging.MessageBus.add_listener()`, `remove_listener()`, `send()`

enabled
Gibt an oder legt fest, ob die Komponente aktiv ist.

Mögliche Werte sind `True` wenn die Komponente aktiv ist und `False` wenn die Komponente nicht aktiv ist.

Alle Komponenten eines Jobs werden beim Aktivieren des Jobs automatisch aktiviert und mit dem Deaktivieren des Jobs automatisch deaktiviert. Das manuelle Setzen dieses Attributs ist i.d.R. nicht notwendig.

Eine aktive Komponente bekommt Geräte (Bricks und Bricklets) zugeordnet und kann Nachrichten empfangen und senden. Eine inaktive Komponente bekommt keine Geräte zugeordnet und erhält auch keine Nachrichten vom Nachrichtensystem zugestellt.

event_tracing
Legt fest, ob über das Nachrichtensystem versendete Nachrichten auf der Konsole protokolliert werden sollen.

Mögliche Werte sind `True` oder `False`.

Siehe auch: `send()`

job
Gibt den Eltern-Job oder `None` zurück.

Siehe auch: `Job.add_component()`

name
Gibt den Namen der Komponente zurück.

on_add_component(*job*)
Wird aufgerufen, wenn die Komponente einem Job hinzugefügt wird.

Parameter

job Eine Referenz auf den Job.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Job.add_component()`

on_core_started()
Wird aufgerufen, wenn der Anwendungskern gestartet wurde.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `Core.start()`

on_core_stopped()
Wird aufgerufen, wenn der Anwendungskern gestoppt wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `Core.stop()`

on_disabled()
Wird aufgerufen, wenn die Komponente deaktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `enabled`

on_enabled()
Wird aufgerufen, wenn die Komponente aktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `enabled`

on_job_activated()
Wird aufgerufen, wenn der Eltern-Job der Komponente aktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `Job.active`, `Core.activate()`

on_job_deactivated()
Wird aufgerufen, wenn der Eltern-Job der Komponente deaktiviert wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden.

Siehe auch: `Job.active`, `Core.deactivate()`

on_remove_component()
Wird aufgerufen, wenn die Komponente aus einem Job entfernt wird.

Bemerkung: Kann von abgeleiteten Klassen überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: `Job.remove_component()`

remove_device_handle(*device_handle*)
Entfernt eine Geräteanforderung.

Als Parameter wird ein `devices.SingleDeviceHandle` oder ein `devices.MultiDeviceHandle` übergeben.

Bemerkung: Es muss die selbe Referenz übergeben werden, wie an `add_device_handle()` übergeben wurde.

Siehe auch: `add_device_handle()`

`remove_listener`(*listener*)
Meldet einen Nachrichtenempfänger vom Nachrichtensystem ab.

Bemerkung: Es muss das selbe Empfängerobjekt übergeben werden, wie an `add_listener()` übergeben wurde.

Siehe auch: `messaging.MessageBus.remove_listener()`, `add_listener()`, `send()`

`send`(*name*, *value=None*)
Versendet eine Nachricht über das Nachrichtensystem.

Parameter

`name` Der Ereignisname für die Nachricht.

`value` ***(optional)*** Der Inhalt der Nachricht. Das kann ein beliebiges Objekt sein.

Beschreibung

Die Methode übergibt die Nachricht an das Nachrichtensystem und kehrt sofort zum Aufrufer zurück. Die Nachricht wird asynchron an die Empfänger übermittelt. Als Absender-Job wird der Name des Eltern-Jobs dieser Komponente eingetragen. Als Absenderkomponente wird der Name dieser Komponente eingetragen.

Siehe auch: `add_listener()`, `remove_listener()`, `messaging.MessageBus.send()`

`trace`(*text*)
Schreibt eine Nachverfolgungsmeldung mit dem Ursprung `Component <Job> <Name>` auf die Konsole.

`tracing`
Legt fest, ob Nachverfolgungsmeldungen für diese Komponente auf der Konsole ausgegeben werden sollen.

Mögliche Werte sind `True` oder `False`.

Siehe auch: `trace()`

4.2 Modul orbit_framework.devices

Dieses Modul implementiert den Gerätemanager von ORBIT.

Das Modul umfasst die folgenden Klassen:

- `DeviceManager`
- `DeviceHandle`
- `SingleDeviceHandle`
- `MultiDeviceHandle`

Darüber hinaus enthält es einige Hilfsfunktionen für den Umgang mit TinkerForge-Gerätetypen:

- `get_device_identifier()`
- `device_identifier_from_name()`
- `device_instance()`
- `device_name()`
- `known_device()`

4.2.1 Klassen

DeviceManager

class orbit_framework.devices.**DeviceManager**(*core*)
Diese Klasse implementiert den Gerätemanager einer ORBIT-Anwendung.

Parameter

core Ein Verweis auf den Anwendungskern der ORBIT-Anwendung. Eine Instanz der Klasse Core.

Beschreibung

Der Gerätemanager baut eine Verbindung zu einem TinkerForge-Server auf, ermittelt die angeschlossenen Bricks und Bricklets und stellt den Komponenten in den Jobs die jeweils geforderten Geräte zur Verfügung.

Dabei behält der Gerätemanager die Kontrolle über den Gerätezugriff. Das bedeutet, dass der Gerätemanager die Autorität hat, einer Komponente ein Gerät zur Verügung zu stellen, aber auch wieder zu entziehen.

Eine Komponente bekommt ein von ihm angefordertes Gerät i.d.R. dann zugewiesen, wenn die Komponente aktiv und das Gerät verfügbar ist. Wird die Verbindung zum TinkerForge-Server unterbrochen oder verliert der TinkerForge-Server die Verbindung zum Master-Brick (USB-Kabel herausgezogen), entzieht der Gerätemanager der Komponente automatisch das Gerät, so dass eine Komponente i.d.R. keine Verbindungsprobleme behandeln muss.

Umgesetzt wird dieses Konzept mit Hilfe der Klassen SingleDeviceHandle und MultiDeviceHandle.

add_device_callback(*uid*, *event*, *callback*)
Richtet eine Callback-Funktion für ein Ereignis eines Bricks oder eines Bricklets ein.

Parameter

uid Die UID des Gerätes für das ein Ereignis abgefangen werden soll.

event Die ID für das abzufangene Ereignis. Z.B. tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.

callback Eine Callback-Funktion die bei Auftreten des Ereignisses aufgerufen werden soll.

Beschreibung

Da jedes Ereignis andere Ereignisparameter besitzt, muss die richtige Signatur für die Callbackfunktion der TinkerForge-Dokumentation entnommen werden. Die Ereignisparameter werden in der API-Dokumentation für jeden Brick und jedes Bricklet im Abschnitt *Callbacks* beschrieben.

Bemerkung: Der Gerätemanager stellt einen zentralen Mechanismus für die Registrierung von Callbacks für Geräteereignisse zur Verfügung, weil die TinkerForge-Geräteklassen nur ein Callback per Ereignis zulassen. Der Gerätemanager hingegen unterstützt beliebig viele Callbacks für ein Ereignis eines Gerätes.

Siehe auch: remove_device_callback()

add_device_finalizer(*device_identifier*, *finalizer*)
Richtet eine Abschlussfunktion für einen Brick- oder Bricklet-Typ ein.

Parameter

device_identifier Die Geräte-ID der TinkerForge-API. Z.B. tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.DEVICE_IDENTIFIER

finalizer Eine Funktion, welche als Parameter eine Instanz der TinkerForge-Geräteklasse entgegennimmt.

Beschreibung

Sobald der Gerätemanager die Verbindung zu einem Gerät selbstständig aufgibt (d.h. die Verbindung nicht durch eine Störung unterbrochen wurde), ruft er alle Abschlussfunktionen für die entsprechende Geräte-ID auf.

Siehe auch: `add_device_initializer()`

add_device_initializer(*device_identifier*, *initializer*)
Richtet eine Initialisierungsfunktion für einen Brick- oder Bricklet-Typ ein.

Parameter

device_identifier Die Geräte-ID der TinkerForge-API. Z.B. `tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.DEVICE_IDENTIFIER`

initializer Eine Funktion, welche als Parameter eine Instanz der TinkerForge-Geräteklasse entgegennimmt.

Beschreibung

Sobald der Gerätemanager ein neues Gerät entdeckt, zu dem er bisher keine Verbindung aufgebaut hatte, ruft er alle Initialisierungsfunktionen für die entsprechende Geräte-ID auf.

Siehe auch: `add_device_finalizer()`

add_handle(*device_handle*)
Richtet eine Geräteanforderung (Geräte-Handle) ein.

Eine Geräteanforderung ist eine Instanz einer Sub-Klasse von `DeviceHandle`. Das kann entweder eine Instanz von `SingleDeviceHandle` oder von `MultiDeviceHandle` sein.

Das übergebene Geräte-Handle wird über alle neu entdeckten Geräte mit einem Aufruf von `DeviceHandle.on_bind_device()` benachrichtigt. Je nach Konfiguration nimmt das Handle das neue Gerät an oder ignoriert es. Verliert der Gerätemanager die Verbindung zu einem Gerät, wird das Geräte-Handle ebenfalls mit einem Aufruf von `DeviceHandle.on_unbind_device()` benachrichtigt.

Siehe auch: `remove_handle()`

devices
Ein Dictionary mit allen zur Zeit verfügbaren Geräten. Die UID des Geräts ist der Schlüssel und der Wert ist eine Instanz der TinkerForge-Geräte-Klasse (wie z.B. `tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4`).

remove_device_callback(*uid*, *event*, *callback*)
Entfernt eine Callback-Funktion von einem Ereignis eines Bricks oder eines Bricklets.

Parameter

uid Die UID des Gerätes für das ein Callback aufgehoben werden soll.

event Die ID für das Ereignis. Z.B. `tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.CALLBACK_`

callback Die registrierte Callback-Funktion die entfernt werde soll.

Beschreibung

Für die Aufhebung des Callbacks muss die gleiche Funktionsreferenz übergeben werden wie bei der Einrichtung des Callback.

Siehe auch: `add_device_callback()`

remove_handle(*device_handle*)
Entfernt eine Geräteanforderung (Geräte-Handle).

Eine Geräteanforderung ist eine Instanz einer Sub-Klasse von `DeviceHandle`. Das kann entweder eine Instanz von `SingleDeviceHandle` oder von `MultiDeviceHandle` sein.

Siehe auch: `add_handle()`

start()
Startet den Gerätemanager und baut eine Verbindung zu einem TinkerForge-Server auf. Die Verbindungdaten für den Server werden der ORBIT-Konfiguration entnommen.

Gibt `True` zurück, wenn die Verbindung aufgebaut werden konnte, sonst `False`.

Siehe auch: `stop()`

stop()
Trennt die Verbindung zum TinkerForge-Server und beendet den Gerätemanager.

Vor dem Trennen der Verbindung wird die Zuordnung zwischen den Geräten und den Komponenten aufgehoben.

Siehe auch: `start()`

trace(*text*)
Schreibt eine Nachverfolgungsmeldung mit dem Ursprung `DeviceManager` auf die Konsole.

DeviceHandle

class `orbit_framework.devices.`**DeviceHandle**(*name*, *bind_callback*, *unbind_callback*)
Diese Klasse ist die Basisklasse für Geräteanforderungen.

Anstelle diese Klasse direkt zu verwenden, werden die beiden Kindklassen `SingleDeviceHandle` und `MultiDeviceHandle` verwendet.

Parameter

name Der Name der Geräteanforderung. Dieser Name hilft, verschiedene Geräte, die von einer Komponente angesteuert werden, zu unterscheiden.

bind_callback Eine Funktion, die aufgerufen wird, sobald die Verbindung zu einem Gerät verfügbar ist.

unbind_callback Eine Funktion, die aufgerufen wird, sobald eine Verbindung zu einem Gerät verloren geht.

Beschreibung

Eine Geräteanforderung repräsentiert die Fähigkeit einer Komponente mit einem Typ von TinkerForge-Brick(let)s zu kommunizieren. Sie entbindet die Komponente aber von der Aufgabe die Verbindung zu den Brick(let)s zu verwalten. Die Komponente wird durch Callbacks darüber informiert, ab wann ein Brick(let) verfügbar ist und ab wann es nicht mehr verfügbar ist. Eine Geräteanforderung kann mehrere Brick(let)s gleichen Typs umfassen. Eine Komponente kann jederzeit alle verfügbaren Brick(let)s einer Geräteanforderung abfragen und ansteuern.

Geräteanforderungen verwalten auch Ereignis-Callbacks für Brick(let)s. Eine Komponente kann durch den Aufruf von `register_callback()` ein Callback für ein Brick(let)-Ereignis einrichten und die Geräteanforderung übernimmt gemeinsam mit dem `DeviceManager` die Verwaltung.

Siehe auch: `SingleDeviceHandle`, `MultiDeviceHandle`, `orbit_framework.Component.add_device_ha`
`register_callback()`

accept_device(*device*)
Nimmt ein Gerät in die Geräteanforderung auf. Wird von `on_bind_device()` aufgerufen.

Siehe auch: `on_bind_device()`

devices
Gibt eine Liste mit allen verfügbaren Brick(let)s zurück.

Ein Brick(let) wird durch ein Objekt der entsprechenden TinkerForge-Geräteklasse (wie z.B. `tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4`) repräsentiert.

for_each_device(*f*)
Führt eine Funktion für alle verfügbaren Geräte dieser Geräteanforderung aus.

name
Gibt den Namen der Geräteanforderung zurück.

on_add_handle(*device_manager*)
Wird aufgerufen, wenn die Geräteanforderung im DeviceManager registriert wurde.

Bemerkung: Kann von einer abgeleiteten Klasse überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: DeviceManager.add_handle(), on_remove_handle()

on_bind_device(*device*)
Wird aufgerufen, wenn ein beliebiges Gerät verfügbar wird.

Bemerkung: Muss von einer abgeleiteten Klasse überschrieben werden.

Eine Implementierung muss die Methode accept_device() für alle Geräte aufrufen, die der Geräteanforderung entsprechen.

Siehe auch: accept_device()

on_remove_handle()
Wird aufgerufen, wenn die Geräteanforderung im DeviceManager deregistriert wurde.

Bemerkung: Kann von einer abgeleiteten Klasse überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: DeviceManager.remove_handle(), on_add_handle()

on_unbind_device(*device*)
Wird aufgerufen, wenn ein beliebiges Gerät nicht mehr verfügbar ist.

Ruft die Methode release_device() auf, wenn das Gerät in dieser Geräteanforderung gebunden ist.

Siehe auch: release_device()

register_callback(*event_code*, *callback*)
Richtet ein Callback für ein Brick(let)-Ereignis ein.

Parameter

event_code Der Code für das Brick(let)-Ereignis. Der Code kann der TinkerForge-Dokumentation entnommen werden. (Z.B. tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.CALLBACK_BUTTON_PRESSED)

callback Das Callback, das bei Eintreten des Ereignisses aufgerufen werden soll. Die Signatur muss dem Ereignis entsprechen und ist der TinkerForge-Dokumentation zu entnehmen.

Beschreibung

Für jeden Ereignis-Code kann in jeder Geräteanforderung immer nur ein Callback registriert werden. Das Callback wird immer aufgerufen, sobald ein verfügbares Brick(let) dieser Geräteanforderung das beschriebene Ereignis auslöst.

Siehe auch: unregister_callback(), DeviceManager.add_device_callback()

release_device(*device*)
Wird aufgerufen, wenn ein Gerät aus der Geräteanforderung nicht mehr verfügbar ist.

Bemerkung: Kann von einer abgeleiteten Klasse überschrieben werden. Eine überschreibende Methode muss jedoch die Implementierung der Elternklasse aufrufen.

Siehe auch: on_unbind_device()

unregister_callback(*event_code*)
Meldet ein Callback von einem Brick(let)-Ereignis ab.

Parameter

event_code Der Code für das Brick(let)-Ereignis.

Siehe auch: register_callback(), DeviceManager.remove_device_callback()

SingleDeviceHandle

class orbit_framework.devices.**SingleDeviceHandle**(*name*, *device_name_or_id*, *bind_callback=None*, *unbind_callback=None*, *uid=None*, *auto_fix=False*)

Eine Geräteanforderung für ein einzelnes Brick(let).

Parameter

name Der Name der Geräteanforderung. Der Name wird zur Unterscheidung von mehreren Geräteanforderungen einer Komponente verwendet.

device_name_or_id Der Typenname oder die Typen-ID des Gerätes. Wird eine Zeichenkette übergeben, wird sie als Name des Brick(let)s interpretiert, z.B. 'LCD 20x4 Bricklet'. Wird eine Zahl übergeben, wird sie als Typen-ID interpretiert, z.B. tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.DEVICE_IDENTIFIER.

bind_callback (*optional*) Ein Callback das aufgerufen wird, sobald ein Gerät an die Geräteanforderung gebunden wird. Der Standardwert ist None.

unbind_callback (*optional*) Ein Callback das aufgerufen wird, sobald ein gebundenes Gerät nicht mehr verfügbar ist. Der Standardwert ist None.

uid (*optional*) Die UID eines Brick(let)s. Wenn eine UID angegeben wird, akzeptiert die Geräteanforderung nur genau dieses Brick(let). Wenn keine UID angegeben wird, wird das erste Gerät mit dem angegebenen Gerätetyp akzeptiert. Der Standardwert ist None.

auto_fix (*optional*) Gibt an, ob nach der Bindung zwischen einem Gerät und der Geräteanforderung andere Geräte gebunden werden dürfen.

Mögliche Werte sind True, wenn nach dem ersten gebundenen Gerät andere Geräte gebunden werden dürfen, und False, wenn nach einer Bindung nur noch dieses Gerät gebunden werden darf. Der Standardwert ist False.

Beschreibung

Diese Variante einer Geräteanforderung akzeptiert zu jeder Zeit immer nur ein Gerät. Wird keine UID angegeben, um ein konkretes Brick(let) zu beschreiben, wird das erste Brick(let) des angegebenen Gerätetyps aktzeptiert. Wird die Verbindung zum gebundenen Gerät getrennt, entscheidet der Parameter auto_fix ob anschließend das nächste verfügbare Gerät mit dem passenden Typ akzeptiert wird oder ob solange gewartet wird, bis das vormals gebundene Brick(let) wieder verfügbar wird.

Siehe auch: DeviceHandle, MultiDeviceHandle

device
Gibt das gebundene Brick(let) oder None zurück.

Ein Brick(let) wird durch ein Objekt der entsprechenden TinkerForge-Geräteklasse (wie z.B. tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4) repräsentiert.

MultiDeviceHandle

class orbit_framework.devices.**MultiDeviceHandle**(*name*, *device_name_or_id*, *bind_callback=None*, *unbind_callback=None*)

Eine Geräteanforderung für alle Brick(let)s eines Typs.

Parameter

name Der Name der Geräteanforderung. Der Name wird zur Unterscheidung von mehreren Geräteanforderungen einer Komponente verwendet.

device_name_or_id Der Typenname oder die Typen-ID des Gerätes. Wird eine Zeichenkette übergeben, wird sie als Name des Brick(let)s interpretiert, z.B. `'LCD 20x4 Bricklet'`. Wird eine Zahl übergeben, wird sie als Typen-ID interpretiert, z.B. `tinkerforge.bricklet_lcd_20x4.BrickletLCD20x4.DEVICE_IDENTIFIER`.

bind_callback (*optional*) Ein Callback das aufgerufen wird, sobald ein Gerät an die Geräteanforderung gebunden wird.

unbind_callback (*optional*) Ein Callback das aufgerufen wird, sobald ein gebundenes Gerät nicht mehr verfügbar ist.

Beschreibung

Diese Variante einer Geräteanforderung akzeptiert alle Geräte des angegebenen Gerätetyps.

Siehe auch: `DeviceHandle`, `SingleDeviceHandle`

4.2.2 Funktionen

`orbit_framework.devices.`**`get_device_identifier`**(*name_or_id*)
Ermittelt die Geräte-ID für einen Gerätetyp. Es kann der Name des Gerätetyps oder die Geräte-ID übergeben werden.

`orbit_framework.devices.`**`device_identifier_from_name`**(*name*)
Gibt die Geräte-ID für einen Namen zurück.

`orbit_framework.devices.`**`device_instance`**(*device_identifier*, *uid*, *ipcon*)
Erzeugt eine TinkerForge-Binding-Instanz anhand der Geräte-ID, der UID und der Verbindung.

`orbit_framework.devices.`**`device_name`**(*device_identifier*)
Gibt den Namen eines Gerätetyps anhand der Geräte-ID zurück.

`orbit_framework.devices.`**`known_device`**(*device_identifier*)
Überprüft ob eine Geräte-ID bekannt ist.

4.3 Modul orbit_framework.index

Dieses Modul unterstützt das Nachrichtensystem von ORBIT mit einer hierarchischen Indexstruktur.

Das Modul enthält die folgenden Klassen:

- `MultiLevelReverseIndex`

4.3.1 MultiLevelReversedIndex

class `orbit_framework.index.`**`MultiLevelReverseIndex`**(*attributes*, *item_attribute_selector = getattr*, *lookup_attribute_selector = getattr*)
Diese Klasse implementiert eine hierarchischen Indexstruktur mit Unterstützung für mehrstufige Schlüssel einschließlich Gruppen und Wildcards im Schlüssel.

Parameter

attributes Eine Liste mit den Namen der Indexattribute.

item_attribut_selector (*optional*) Eine Funktion die beim Hinzufügen eines Objektes mit `add()` verwendet wird, um den Attributwert unter Angabe des Objektes und des Attributnamens zu ermitteln. Standardwert ist `getattr()`.

lookup_attribut_selector (*optional*) Eine Funktion die beim Look-Up mit `lookup()` verwendet wird, um den Attributwert unter Angabe des Schlüsselobjektes und des Attributnamens zu ermitteln. Standardwert ist `getattr()`.

Beschreibung

Der Index wird mit einer Liste von Attributnamen (Indexattribute) initialisiert. Jedes Indexattribut steht für eine Ebene des hierarchischen Index.

Gruppen für jedes Indexattribut können mit `add_group()` und `delete_group()` verwaltet werden.

Objekte werden mit `add()` hinzugefügt und anhand der Indexattribute indiziert. Die Attributwerte des Objektes können konkrete Werte, ein Gruppennamen oder *None* als Wildcard sein.

Ein Look-Up erfolgt mit `lookup()` und der Angabe eines Schlüsselobjekts. Ein Schlüsselobjekt besitzt wie die indizierten Objekte alle Indexattribute. Zurückgegeben werden bei einem Look-Up alle Objekte, deren Attributwerte zum angegebenen Schlüsselobjekt passen.

add(*item*)
Fügt dem Index ein Objekt hinzu. Das Objekt wird anhand der Indexattribute indiziert.

Die Werte der Indexattribute können konkrete Werte, Gruppennamen oder *None* als Wildcard sein.

add_group(*attribute*, *group*, *keys*)
Fügt dem Index eine Gruppe für ein Attribut hinzu.

attribute Der Name des Indexattributes für das eine Gruppe eingerichtet werden soll.

group Der Gruppenname unter dem Objekte indiziert werden können.

keys Eine Sequenz von Werten die bei Look-Ups zu Objekten führen welche unter dem Gruppennamen indiziert wurden.

delete_group(*attribute*, *group*)
Entfernt eine Gruppe für ein Attribut aus dem Index.

attribute Der Name des Indexattributes für das die Gruppe entfernt werden soll.

group Der Name der Gruppe.

is_empty()
Gibt `True` zurück, wenn der Index kein Objekt enthält, sonst `False`.

Der Index ist auch dann leer, wenn Gruppen eingerichtet, aber keine Objekte indiziert wurden.

lookup(*key_obj*)
Ruft eine Liste mit allen Objekten ab, deren indizierte Attributwerte zu den Attributwerten des übergebenen Schlüsselobjektes passen.

Damit ein Objekt im Ergebnis enthalten ist, müssen alle Indexattribute passen. Ein Indexattribut passt

1. wenn der indizierte Attributwert gleich dem Schlüsselattribut ist
2. oder das indizierte Attribut ein Gruppenname und das Schlüsselattribut in dieser Gruppe ist,
3. oder das indizierte Attribut *None* ist.

remove(*item*)
Entfernt ein Objekt aus dem Index.

4.4 Modul orbit_framework.messaging

Dieses Modul implementiert das Nachrichtensystem von ORBIT.

Das Modul umfasst die folgenden Klassen:

- `MessageBus`
- `Slot`

- Listener
- MultiListener

4.4.1 MessageBus

class orbit_framework.messaging.**MessageBus**(*core*)
Diese Klasse implementiert das ORBIT-Nachrichtensystem.

Parameter

core Ein Verweis auf den Anwendungskern der ORBIT-Anwendung. Eine Instanz der Klasse Core.

Beschreibung

Das Nachrichtensystem funktioniert nach dem Broadcast-Prinzip. Nachrichten werden von einem Absender an das Nachrichtensystem übergeben, ohne dass der Absender weiß, wer die Nachricht empfangen wird (send()).

Empfänger können sich mit einem Empfangsmuster (Slot) bei dem Nachrichtensystem registrieren (add_listener()) und bekommen alle Nachrichten zugestellt, die ihrem Empfangsmuster entsprechen.

Das Zustellen der Nachrichten (der Aufruf der Empfänger-Callbacks) erfolgt in einem dedizierten Thread. Das hat den Vorteil, dass das Versenden einer Nachricht ein asynchroner Vorgang ist, der den Absender nicht blockiert.

Jede Nachricht ist einem Ereignis zugeordnet. Dieses Ereignis ist durch einen Job, eine Komponente und einen Ereignisnamen definiert. Der Absender eines Ereignisses gibt beim Versenden einer Nachricht den Namen des Ereignisses an.

Um das Erzeugen von Empfangsmustern (Slot) zu vereinfachen, können Jobs, Komponenten und Ereignisnamen zu Gruppen zusammengefasst werden (job_group(), component_group(), name_group()).

class **Message**(*job*, *component*, *name*, *value*)
Diese Klasse repräsentiert eine Nachricht im Nachrichtensystem. Sie wird nur intern verwendet.

MessageBus.**add_listener**(*listener*)
Registriert einen Empfänger mit einem Empfangsmuster im Nachrichtensystem.

Das Empfangsmuster besteht aus den drei Attributen job, component und name. Diese Attribute können jeweils einen Namen, einen Gruppennamen oder None enthalten. Sie werden benutzt, um zu entscheiden, ob eine Nachricht an den Empfänger übergeben wird oder nicht.

Üblichweise wird als Empfänger ein Listener-Objekt verwendet, welches mit einem Slot-Objekt und einem Callback initialisiert wurde.

Die Nachricht *M*: job = 'A', component = '1', name = 'a', value = ...
wird an alle Empfänger mit dem Empfangsmuster
⟨ job = 'A', component = '1', name = 'a' ⟩ übergeben.

Existiert eine Ereignisnamengruppe mit dem Namen 'x' und den Ereignisnamen 'a', b', 'c', wird die Nachricht *M* auch an alle Empfänger mit dem Empfangsmuster
⟨ job = 'A', component = '1', name = 'x' ⟩ übergeben.

Hat ein Attribut den Wert None, wird es ignoriert. Das bedeutet, die Nachricht *M* wird auch an alle Empfänger mit dem Empfangsmuster
⟨ job = 'A', component = '1', name = None ⟩ übergeben.

Der Empfänger muss den Aufruf als Funktion unterstützen. Er wird für die Übergabe der Nachricht mit den folgenden vier Parametern aufgerufen:

job Der Name des versendenden Jobs

component Der Name der versendenden Komponente

name Der Name des Ereignisses

value Der Nachrichteninhalt

Die Parameter werden in der dargestellten Reihenfolge als Positionsparameter übergeben.

Siehe auch: `Slot`, `Listener`, `remove_listener()`, `send()`

`MessageBus.`**`component_group`**(*group_name*, **names*)
Richtet eine Absendergruppe auf Komponentenebene ein.

Parameter

group_name Der Name der Gruppe.

names (*var args*) Die Namen der Komponenten, welche in der Gruppe zusammengefasst werden sollen.

Beschreibung

Durch eine Absendergruppe auf Komponentenebene kann ein Slot anstelle eines spezifischen Komponentennamens einen Gruppennamen im Empfangsmuster angeben.

Siehe auch: `Slot`, `Listener`, `add_listener()`, `remove_listener()`, `orbit_framework.Job.add_listener()`, `orbit_framework.Component.add_listener()`

`MessageBus.`**`job_group`**(*group_name*, **names*)
Richtet eine Absendergruppe auf Job-Ebene ein.

Parameter

group_name Der Name der Gruppe.

names (*var args*) Die Namen der Jobs, welche in der Gruppe zusammengefasst werden sollen.

Beschreibung

Durch eine Absendergruppe auf Job-Ebene kann ein Slot anstelle eines spezifischen Jobnamens einen Gruppennamen im Empfangsmuster angeben.

Siehe auch: `Slot`, `Listener`, `add_listener()`, `remove_listener()`, `orbit_framework.Job.add_listener()`, `orbit_framework.Component.add_listener()`

`MessageBus.`**`name_group`**(*group_name*, **names*)
Richtet eine Absendergruppe auf Ereignisebene ein.

Parameter

group_name Der Name der Gruppe.

names (*var args*) Die Namen der Ereignisse, welche in der Gruppe zusammengefasst werden sollen.

Beschreibung

Durch eine Absendergruppe auf Ereignisebene kann ein Slot anstelle eines spezifischen Ereignisnamens einen Gruppennamen im Empfangsmuster angeben.

Siehe auch: `Slot`, `Listener`, `add_listener()`, `remove_listener()`, `orbit_framework.Job.add_listener()`, `orbit_framework.Component.add_listener()`

`MessageBus.`**`remove_listener`**(*listener*)
Entfernt einen Empfänger und sein Empfangsmuster aus dem Nachrichtensystem.

Bemerkung: Es muss das gleiche Objekt übergeben werden, wie an `add_listener()` übergeben wurde.

Siehe auch: `add_listener()`

`MessageBus.`**`send`**(*job*, *component*, *name*, *value*)
Sendet eine Nachricht über das Nachrichtensystem.

Parameter

`job` Der versendende Job

`component` Die versendende Komponente

`name` Der Ereignisname

`value` Der Inhalt der Nachricht

Beschreibung

Die Nachricht wird in die Warteschlange eingestellt. Der Aufruf kehrt sofort wieder zum Aufrufer zurück.

Siehe auch: `add_listener()`

`MessageBus.`**`start`**`()`
Startet das Nachrichtensystem.

Zur Weiterleitung der Nachrichten wird ein dedizierter Thread gestartet.

Siehe auch: `stop()`

`MessageBus.`**`stop`**(*immediate=False*)
Beendet das Nachrichtensystem.

Parameter

`immediate` **(*optional*)** `True` wenn wartende Nachrichten nicht mehr abgearbeitet werden sollen, `False` wenn erst alle wartenden Nachrichten abgearbeitet werden sollen.

Blockiert solange bis der dedizierte Thread für die Nachrichtenverteilung beendet wurde und kehrt erst anschließend zum Aufrufer zurück.

Siehe auch: `start()`

`MessageBus.`**`trace`**(*text*)
Schreibt eine Nachverfolgungsmeldung mit dem Ursprung `MessageBus` auf die Konsole.

4.4.2 Slot

class `orbit_framework.messaging.`**`Slot`**(*job*, *component*, *name*, *predicate=None*, *transformation=None*)
Diese Klasse repräsentiert ein Empfangsmuster für das Nachrichtensystem.

Parameter

`job` Der Name des versendenen Jobs ein Gruppenname oder `None`.

`component` Der Name der versendenen Komponente, `'JOB'`, ein Gruppenname oder `None`.

`name` Der Ereignisname, ein Gruppenname oder `None`.

`predicate` **(*optional*)** Ein Prädikat als zusätzlicher Filter für Nachrichten. Eine Funktion mit der Signatur `(job, component, name, value)`, welche die Nachricht entgegennimmt und `True` oder `False` zurückgibt.

`transformation` **(*optional*)** Eine Funktion, welche den Inhalt der Nachricht entgegennimmt und einen umgewandelten Inhalt zurückkgibt.

Beschreibung

Das Empfangsmuster wird verwendet, um ein Callback für den Nachrichtenempfang im Nachrichtensystem zu registrieren. Das Empfangsmuster kann durch einen Aufruf der Methode `listener()` mit einem Callback zu einem Empfänger verknüpft werden.

Für die Erzeugung von `Slot`-Instanzen gibt es einige statische Factory-Methoden: `for_job()`, `for_component()` und `for_name()`.

Siehe auch: `listener()`, `Listener`

component
Gibt den Namen der versendenden Komponente, `'JOB'`, einen Gruppennamen oder `None` zurück.

for_component(*job*, *component*)
Erzeugt ein Empfangsmuster, welches auf die Nachrichten aller Komponenten mit dem übergegebenen Namen passt. Es kann auch ein Gruppenname übergegen werden.

for_job(*job*)
Erzeugt ein Empfangsmuster, welches auf alle Nachrichten von einem Job passt. Es kann auch ein Gruppenname übergegen werden.

for_name(*name*)
Erzeugt ein Empfangsmuster, welches auf alle Nachrichten für das übergebene Ereignis passt. Es kann auch ein Gruppenname übergegen werden.

job
Gibt den Namen des versendenden Jobs, einen Gruppennamen oder `None` zurück.

listener(*callback*)
Erzeugt mit dem übergebenen Callback einen Empfänger.

Siehe auch: `Listener`

name
Gibt den Ereignisnamen, einen Gruppenname oder `None` zurück.

predicate
Gibt das Filterprädikat oder `None` zurück.

transformation
Gibt die Transformationsfunktion für den Nachrichteninhalt oder `None` zurück.

4.4.3 Listener

class `orbit_framework.messaging.`**Listener**(*callback*, *slot*)
Diese Klasse repräsentiert einen Empfänger für das Nachrichtensystem.

Parameter

callback Das Callback welches beim Eintreffen einer passenden Nachricht aufgerufen werden soll. Die Funktion muss die folgende Signatur besitzen: `(job, component, name, value)`. Wird eine dynamische Methode statt einer statischen Funktion übergeben, muss die Methode die folgende Signatur besitzen: `(self, job, component, name, value)`.

slot Ein Empfangsmuster für die Filterung der Nachrichten.

Ein Empfänger kann mit Hilfe der folgenden Methoden für den Nachrichtenempfang registriert werden: `MessageBus.add_listener()`, `orbit_framework.Job.add_listener()` und `orbit_framework.Component.add_listener()`.

Siehe auch: `Slot`, `MessageBus`

component
Gibt den Namen der versendenden Komponente, `'JOB'`, einen Gruppennamen oder `None` zurück.

job
Gibt den Namen des versendenden Jobs, einen Gruppennamen oder `None` zurück.

name
Gibt den Ereignisnamen, einen Gruppenname oder `None` zurück.

receiver
Gibt die Bezeichnung des Empfängers zurück.

Die Bezeichnung wird beim Protokollieren der Nachrichten verwendet, um den Weg der Nachrichten kenntlich zu machen.

Werden die Methoden `orbit_framework.Job.add_listener()` oder `orbit_framework.Component.add_listener()` verwendet, wird dieses Attribut automatisch gesetzt. Wird der Empfänger direkt mit `MessageBus.add_listener()` registriert, sollte dieses Attribut vorher gesetzt werden, um den Empfänger zu bezeichnen.

4.4.4 MultiListener

class `orbit_framework.messaging.`**MultiListener**(*name*, *callback*)
Diese Klasse implementiert einen Mehrfachempfänger. Das ist ein Mechanismus für den Empfang von Nachrichten über das Nachrichtensystem, bei dem mehrere Empfangsmuster mit einem Callback verknüpft werden.

Parameter

name Der Name des Empfängers.

callback Eine Funktion die bei einem Nachrichtenempfang aufgerufen werden soll. Die Funktion muss die Signatur `(job, component, name, value)` besitzen.

Beschreibung

Der Mehrfachempfänger wird mit einem Namen und einem Callback initialisiert. Anschließend können mit `add_slot()` mehrere Empfangsmuster eingerichtet werden.

Bemerkung: Der Mehrfachempfänger erzeugt für jedes Empfangsmuster einen eigenen Empfänger. Passt eine Nachricht zu mehr als einem Empfangsmuster, wird das Callback für die Nachricht auch mehr als einmal aufgerufen.

Der Mehrfachempfänger muss mit dem Nachrichtensystem verknüpft werden, damit er funktioniert. Dazu wird die Methode `activate()` aufgerufen.

Siehe auch: `Listener`, `add_slot()`, `activate()`

activate(*message_bus*)
Verknüpft den Mehrfachempfänger mit dem Nachrichtensystem.

add_slot(*slot*)
Fügt ein Empfangsmuster hinzu.

Siehe auch: `Slot`

deactivate(*message_bus*)
Löst die Verbindung des Mehrfachempfängers vom Nachrichtensystem.

listeners
Gibt eine Sequenz mit allen zur Zeit erzeugten Empfängern zurück.

name
Gibt den Namen des Empfängers zurück.

remove_slot(*slot*)
Entfernt ein Empfangsmuster.

Bemerkung: Es muss die selbe Referenz auf das Empfangsmuster übergeben werden, wie an

add_slot() übergeben wurde.

slots
Gibt eine Sequenz mit allen eingerichteten Empfangsmustern zurück.

4.5 Modul orbit_framework.setup

Dieses Modul implementiert die Setup-Steuerung für ORBIT.

Bemerkung: Dieses Modul kann auch als Skript ausgeführt werden, es gibt dann die aktuelle ORBIT-Konfiguration aus.

Dieses Modul enthält die folgenden Klassen:

- Configuration

4.5.1 Configuration

class orbit_framework.setup.**Configuration**
Diese Klasse verwaltet globale Konfigurationsparameter für ORBIT. Jeder Parameter ist mit einem Standardwert vorbelegt und kann in der Datei <Benutzerverzeichnis>/.orbit überschrieben werden. Dazu besitzt die Kasse die Methoden load() and save(). Die Methode load() wird bereits automatisch beim Instanzieren aufgerufen.

Die Klasse unterstützt die String-Darstellung mit str().

component_tracing
Aktiviert die Trace-Nachrichten auf Component-Ebene. (*bool*)

connection_retry_time
Die Zeitspanne in Sekunden nachdem eine fehlgeschlagene IP-Verbindung wieder aufgebaut werden soll. (*int*)

core_tracing
Aktiviert die Trace-Nachrichtenausgabe vom ORBIT-Kern. (*bool*)

device_tracing
Aktiviert die Trace-Nachrichten des Gerätemanagers von ORBIT. (*bool*)

event_tracing
Aktiviert die Trace-Nachrichten des Nachrichtenbusses von ORBIT. (*bool*)

host
Der Hostname für die Netzwerkverbindung zum Brick-Deamon oder Master-Brick. (*string*)

job_tracing
Aktiviert die Trace-Nachrichten auf Job-Ebene. (*bool*)

load()
Diese Method lädt die überschrieben Parameterwerte aus der benutzerspezifischen Konfigurationsdatei.

port
Der Port für die Netzwerkverbindung zum Brick-Deamon oder Master-Brick. (*int*)

save()
Speichert die Werte aller Parameter in die benutzerspezifische Konfigurationsdatei.

4.6 Modul orbit_framework.tools

Diese Modul enthält unterstützende Klassen und Funktionen.

Das Modul enthält die folgenden Klassen:

- `MulticastCallback`

4.6.1 MulticastCallback

class `orbit_framework.tools.`**`MulticastCallback`**

Diese Klasse bildet einen einfachen Mechanismus, um mehrere Callbacks mit identischer Signatur zu einem Callback zusammenzufassen.

Mit `add_callback()` können Callbacks hinzugefügt werden. Mit `remove_callback()` können Callbacks wieder entfernt werden.

Die Klasse implementiert die `__call__`-Methode, daher können Instanzen der Klasse selbst als Callback weitergegeben werden. Wird Klasse als Funktion aufgerufen, werden die mit `add_callback()` registrierten Funktionen in der gleichen Reihenfolge aufgerufen, in der sie hinzugefügt wurden. Dabei werden alle Parameter unverändert weitergegeben.

Siehe auch: `add_callback()`, `remove_callback()`

`add_callback`(*callback*)

Fügt ein Callback hinzu.

`remove_callback`(*callback*)

Entfernt ein Callback.

Integrierte Komponenten

ORBIT bringt einige integrierte Komponenten mit. Diese gliedern sich in die folgenden Bereiche:

- Common
- Timer
- LCD
- Remote Switch

5.1 Common

Dieses Modul enthält einige Komponenten für den allgemeinen Einsatz.

Enthalten sind die folgenden Komponenten:

- `EventCallbackComponent`

5.1.1 EventCallbackComponent

class `orbit_framework.components.common.`**`EventCallbackComponent`**(*name*, *slot*, *callback*, ***nargs*)

Diese Komponente wartet mit Hilfe eines Empfangsmusters auf Nachrichten, und ruft ein Callback auf, wenn eine passende Nachricht eintrifft.

Parameter

`name` Der Name der Komponente.

`slot` Das Empfangsmuster für den Empfang der Nachrichten.

`callback` Eine parameterlose Funktion.

5.2 Timer

Dieses Modul enthält einige Komponenten für die zeitabhängige Steuerung.

Enthalten sind die folgenden Komponenten:

- `ActivityTimerComponent`
- `IntervalTimerComponent`

5.2.1 ActivityTimerComponent

class orbit_framework.components.timer.**ActivityTimerComponent**(*name*, *slot*, *initial_state=True*, *timeout=6*, ***nargs*)

Diese Komponente schaltet zwischen den zwei Zuständen **aktiv** und **inaktiv** um und versendet Nachrichten bei einer Änderung des Zustands. Der Zustand wird durch Nachrichtenaktivität gesteuert. Dazu empfängt die Komponente Nachrichten mit einem Empfangsmuster und schaltet auf *aktiv*, sobald eine Nachricht eintrifft. Vergeht eine vorgegebene Zeitspanne ohne eingehende Nachrichten, schaltet die Komponente auf *inaktiv*.

Parameter

name Der Name der Komponente.

slot Das Empfangsmuster für den Empfang der Nachrichten.

initial_state (***optional***) Der Anfangszustand der Komponente. Mögliche Werte sind `True` für *aktiv* und `False` für *inaktiv*. Standardwert ist `True` für *aktiv*.

timeout Die Zeitspanne in Sekunden ohne Nachrichten, bis die Komponente in den Zustand *inaktiv* wechselt.

Nachrichten

Wenn die Komponente in den Zustand *aktiv* wechselt, werden die folgenden beiden Nachrichten versandt:

- *name:* `'state'`, *value:* `True`
- *name:* `'on'`, *value:* `None`

Wenn die Komponente in den Zustand *inaktiv* wechselt, werden die folgenden beiden Nachrichten versandt:

- *name:* `'state'`, *value:* `False`
- *name:* `'off'`, *value:* `None`

5.2.2 IntervalTimerComponent

class orbit_framework.components.timer.**IntervalTimerComponent**(*name*, *interval=1*, ***nargs*)

Diese Komponente implementiert einen Zeitgeber, der in regelmäßigen Abständen eine Nachricht versendet.

Parameter

name Der Name der Komponente.

interval Das Interval für den Nachrichtenversand in Sekunden.

Nachrichten

Die Komponente sendet im angegebenen Intervall die folgende Nachricht:

- *name:* `'timer'`, *value:* Der Zeitpunkt an dem die Nachricht versandt wurde.

5.3 LCD

Dieses Modul enthält einige Komponenten für die Steuerung von LCD-Displays.

Enthalten sind die folgenden Komponenten:

- LCD20x4ButtonsComponent
- LCD20x4BacklightComponent
- LCD20x4WatchComponent

- LCD20x4MessageComponent
- LCD20x4MenuComponent

5.3.1 LCD20x4ButtonsComponent

class orbit_framework.components.lcd.**LCD20x4ButtonsComponent**(*name*, ***nargs*)

Diese Komponente reagiert auf Tastendruck an allen angeschlossenen LCD-20x4-Displays und versendet entsprechende Nachrichten.

Parameter

name Der Name der Komponente.

Nachrichten

Wenn eine Taste gedrückt wird, versendet die Komponente die folgende Nachricht:

name: `'button_pressed'`, *value:* `(uid, no)`

Wenn eine Taste losgelassen wird, versendet die Komponente die folgende Nachricht:

name: `'button_released'`, *value:* `(uid, no)`

Der Nachrichteninhalt ist jeweils ein Tupel aus UID des LCD-Displays und der Tastennummer.

5.3.2 LCD20x4BacklightComponent

class orbit_framework.components.lcd.**LCD20x4BacklightComponent**(*name*, *slot*, *initial_state=False*, ***nargs*)

Diese Komponente schaltet die Hintergrundbeleuchtung aller angeschlossener LCD-20x4-Displays entsprechend dem Nachrichteninhalt eintreffender Nachrichten.

Parameter

name Der Name der Komponente.

slot Das Empfangsmuster für den Nachrichtenempfang.

initial_state (***optional***) Der Anfangszustand der Hintergrundbeleuchtung. Mögliche Werte sind `True` für eingeschaltet und `False` für ausgeschaltet. Der Standardwert ist `False`.

Beschreibung

Trifft eine Nachricht ein, die dem abgegebenen Empfangsmuster entspricht, wird der Nachrichteninhalt entsprechend der Python-Semantik als Wahrheitswert interpretiert. Ist der Wert Wahr, wird die Hintergrundbeleuchtung für aller angeschlossenen LCD-20x4-Displays eingeschaltet, andernfalls wird sie ausgeschaltet.

5.3.3 LCD20x4WatchComponent

class orbit_framework.components.lcd.**LCD20x4WatchComponent**(*name*, *slot*, *lcd_uid=None*, *lines={0: '%d.%m.%Y %H:%M:%S'}*, ***nargs*)

Diese Komponente zeigt beim Eintreffen einer Nachricht auf einem oder allen LCD-20x4-Displays eine formatierte Uhrzeit an.

Parameter

name Der Name der Komponente.

slot Das Empfangsmuster für den Nachrichtenempfang.

lcd_uid (*optional*) Die UID eines LCD-20x4-Displays oder `None`. Wird `None` angegeben, wird die Uhrzeit auf allen angeschlossenen Displays angezeigt. Der Standardwert ist `None`.

lines (*optional*) Ein Dictionary welches eine Zeilennummer auf eine Formatierungszeichenkette abbildet. Die Zeilennummer gibt die 0-basierte Zeile im LCD-Display an. Die Formatierungszeichenkette muss der Formatierungssyntax von `datetime.strftime()` entsprechen. Standardwert ist `{0: '%d.%m.%Y %H:%M:%S'}`.

Beschreibung

Sobald eine Nachricht empfangen wird, die dem angegebenen Empfangsmuster entspricht, wird die aktuelle Uhrzeit mit dem im Parameter `lines` angegebenen Format formatiert angezeigt. Wenn für Parameter `lcd_uid` eine UID übergeben wird, wird die Uhrzeit nur auf dem Display mit dieser UID angezeigt, andernfalls wird die Uhrzeit auf allen angeschlossenen LCD-20x4-Displays angezeigt.

5.3.4 LCD20x4MessageComponent

class `orbit_framework.components.lcd.`**LCD20x4MessageComponent**(*name*, *lines*, *lcd_uid=None*, ***nargs*)

Diese Komponente zeigt eine konstante Nachricht auf einem oder allen LCD-20x4-Displays an.

Parameter

name Der Name der Komponente.

slot Das Empfangsmuster für den Nachrichtenempfang.

lcd_uid (*optional*) Die UID eines LCD-20x4-Displays oder `None`. Wird `None` angegeben, wird die Nachricht auf allen angeschlossenen Displays angezeigt. Der Standardwert ist `None`.

lines Eine Sequenz mit Zeichenketten. Es werden maximal 4 Zeichenketten ausgegeben. Ist eine Zeichenkette länger als 20 Zeichen, wird der Rest abgeschnitten.

Beschreibung

Sobald eine Nachricht empfangen wird, die dem angegebenen Empfangsmuster entspricht, wird die Nachricht aus dem Parameter `lines` angezeigt. Wenn für Parameter `lcd_uid` eine UID übergeben wird, wird die Nachricht nur auf dem Display mit dieser UID angezeigt, andernfalls wird die Nachricht auf allen angeschlossenen LCD-20x4-Displays angezeigt.

5.3.5 LCD20x4MenuComponent

class `orbit_framework.components.lcd.`**LCD20x4MenuComponent**(*name*, *lcd_uid=None*, *entries=[('None', 'none')]*, ***nargs*)

Diese Komponente stellt ein Menü mit bis zu 8 Menüpunkten auf einem LCD-20x4-Display zur Verfügung. Die Navigation im Menü erfolgt mit Hilfe der 4 Tasten des LCD-20x4-Bricklets. Wird ein Menüpunkt ausgewählt, wird eine Nachricht mit der ID des gewählten Menüpunktes versandt.

Parameter

name Der Name der Komponente.

lcd_uid (*optional*) Die UID eines LCD-20x4-Displays oder `None`.

entries (*optional*) Eine Sequenz von Menüpunkten. Jeder Menüpunkt wird durch ein Tuple aus zwei Zeichenketten definiert. Die erste Zeichenkette ist die Beschriftung des Menüpunktes und die zweite Zeichekette ist die ID des Menüpunktes. Der Standardwert ist `[('None', 'none')]`.

Beschreibung

Wenn für den Parameter `lcd_uid` eine UID übergeben wird, wird das Menü auf dem Display mit der angegebenen UID angezeigt, andernfalls wird es auf dem ersten angeschlossenen LCD-20x4-Display angezeigt.

Die Navigation im Menü erfolgt mit den vier Tasten am Display. Die Tasten sind von links nach rechts zwischen 0 und 3 durchnummeriert. Die Tasten sind wie folgt belegt:

0.Escape oder Verlassen - sendet eine Nachricht mit dem Ereignisnamen `'escape'`.

1.Zurück - markiert den vorhergehenden Menüpunkt.

2.Vorwärts - markiert den nächsten Menüpunkt.

3.Enter oder Auswählen - sendet eine Nachricht mit der ID des Menüpunkts als Ereignisname.

Nachrichten

Wenn die Taste 0 gedrückt wird, sendet die Komponente die folgende Nachricht:

- *name:* `'escape'`, *value:* `None`

Wenn die Taste 3 gedrückt wird, sendet die Komponente die folgende Nachricht:

- *name:* `'<ID>'`, *value:* `None`

Wobei `<ID>` durch die ID des aktuell markierten Menüpunkts ersetzt wird.

5.4 Remote Switch

Dieses Modul enthält Komponenten für die Steuerung mit einem Remote-Switch-Bricklet.

Enthalten sind die folgenden Komponenten:

- `RemoteSwitchComponent`

5.4.1 RemoteSwitchComponent

class `orbit_framework.components.remoteswitch.`**`RemoteSwitchComponent`**(*name*, *group*, *socket*, *on_slot*, *off_slot*, *typ='A'*, *switch_uid=None*, ***nargs*)

Diese Komponente steuert eine Funksteckdose mit Hilfe des Remote-Switch-Bricklets wenn Nachrichten über das Nachrichtensystem empfangen werden.

Parameter

`name` Der Name der Komponente.

`group` Die Gruppennummer der Funksteckdose.

`socket` Die ID der Funksteckdose.

`on_slot` Ein Empfangsmuster welches die Funksteckdose einschalten soll.

`on_off` Ein Emfpangsmuster welches die Funksteckdose ausschalten soll.

`typ` (***optional***) Der Typ der Funksteckdose. Mögliche Werte sind `'A'`, `'B'` oder `'C'`. Der Standardwert ist `'A'`. Mehr Informationen zu Steckdosentypen sind in der Remote-Switch-Dokumentation[1] zu finden.

`switch_uid` (***optional***) Die UID des Remote-Switch-Bricklets oder `None` für den ersten der gefunden wird. Der Standardwert ist `None`.

[1] http://www.tinkerforge.com/de/doc/Hardware/Bricklets/Remote_Switch.html

Beschreibung

Wenn eine Nachricht mit dem Empfangsmuster von `on_slot` eintrifft, wird der Einschaltbefehl an die angegebene Steckdose gesendet. Wenn eine Nachricht mit dem Empfangsmuster von `off_slot` eintrifft, wird der Ausschaltbefehl an die angegebene Steckdose gesendet.

Siehe auch: Remote-Switch-Dokumentation[2]

[2] http://www.tinkerforge.com/de/doc/Hardware/Bricklets/Remote_Switch.html

KAPITEL 6

Integrierte Jobs

ORBIT bringt einige integrierte Apps und Services mit.

6.1 Apps

Dieses Modul enthält einige Apps und App-Basisklassen für den allgemeinen Einsatz.

Die folgenden Apps sind enthalten:

- `EscapableApp` (Basisklasse)
- `WatchApp`
- `MessageApp`
- `MenuApp`

6.1.1 EscapableApp

class `orbit_framework.jobs.apps.`**`EscapableApp`**(**args*, ***nargs*)

Dies ist eine Basisklasse für Apps, welche sich durch das Drücken der ersten Taste (ganz links) eines LCD-20x4-Displays selbst beenden.

Diese Funktionalität wird häufig in menügesteuerten Anwendungen im Zusammenspiel mit einem oder mehreren LCD-20x4-Displays benötigt.

Siehe auch: `orbit_framework.components.lcd.LCD20x4ButtonsComponent`

6.1.2 WatchApp

class `orbit_framework.jobs.apps.`**`WatchApp`**(*name*, ***nargs*)

Diese App zeigt, solange sie aktiviert ist, das Datum und die Uhrzeit auf allen angeschlossenen LCD-20x4-Displays an.

Parameter

`name` Der Name der App.

Beschreibung

Diese App erbt von `EscapableApp` und kann daher mit einem Druck auf die erste Taste eines LCD-Displays (ganz links) deaktiviert werden.

Siehe auch: `orbit_framework.components.lcd.LCD20x4WatchComponent`, `orbit_framework.components.timer.IntervalTimerComponent`

6.1.3 MessageApp

class orbit_framework.jobs.apps.**MessageApp**(*name*, *lines*, ***nargs*)

Diese App zeigt, solange sie aktiviert ist, eine vorgegebene Nachricht auf allen angeschlossenen LCD-20x4-Displays an.

Parameter

name Der Name der App.

lines Eine Sequenz mit bis zu vier Zeichenketten. Jede Zeichenkette wird in einer der vier LCD-Zeilen dargestellt. Ist eine Zeichenkette länger als 20 Zeichen, wird der Rest abgeschnitten.

Beschreibung

Diese App erbt von EscapableApp und kann daher mit einem Druck auf die erste Taste eines LCD-Displays (ganz links) deaktiviert werden.

Siehe auch: orbit_framework.components.lcd.LCD20x4MessageComponent

6.1.4 MenuApp

class orbit_framework.jobs.apps.**MenuApp**(*name*, *entries*, ***nargs*)

Diese App zeigt, solange sie aktiviert ist, ein Menü auf dem ersten angeschlossenen LCD-20x4-Display an.

Parameter

name Der Name der App.

entries Eine Sequenz mit Menüpunkten. (*siehe auch:* orbit_framework.components.lcd.LCD20x4MenuComp

Beschreibung

Diese App wird durch die *escape*-Nachricht der Menü-Komponente deaktiviert. Sie kann also durch einen Druck auf die erste Taste (ganz links) des LCD-Displays deaktiviert werden.

Bemerkung: Das Attribut orbit_framework.App.in_history wird für diese App standardmäßig auf True gesetzt. In Folge wird die Menü-App in der App-History vermerkt und es kann unkompliziert durch mehrere verknüpfte Menü-Apps navigiert werden möglich.

Siehe auch: orbit_framework.components.lcd.LCD20x4MenuComponent

6.2 Services

Dieses Modul enthält einige Services für den allgemeinen Einsatz.

Enthalten sind die folgenden Services:

- StandbyService

6.2.1 StandbyService

class orbit_framework.jobs.services.**StandbyService**(*name*, *activity_slot*, *timeout=6*, ***nargs*)

Dieser Service überwacht mit Hilfe eines Empfangsmusters die Benutzeraktivität und schaltet die Anwendung nach einem vorgegebenen Zeitfenster ohne Aktivität in den Standby-Modus. Was soviel bedeutet, wie die Hintergrundbeleuchtung der angeschlossenen LCD-Displays abzuschalten und die App-History zu leeren.

Parameter

name Der Name des Service.

activity_slot Ein Empfangsmuster für alle Nachrichten, die als Benutzeraktivität interpretiert werde können.

timeout (***optional***) Das Länge des Zeitfensters für Inaktivität in Sekunden, nach dem in den Standby geschaltet wird. Standardwert ist `6` Sekunden.

Nachrichten

Der Service sendet die folgenden Nachrichten, wenn Benutzeraktivität detektiert wird:

- *component:* `'standby_timer'`, *name:* `'state'`, *value:* `True`
- *component:* `'standby_timer'`, *name:* `'on'`, *value:* `None`

Wenn die angegebene Zeitspanne für Inaktivität erreicht ist, wird in den Standby-Modus geschaltet und es werden die folgenden Nachrichten versandt:

- *component:* `'standby_timer'`, *name:* `'state'`, *value:* `False`
- *component:* `'standby_timer'`, *name:* `'off'`, *value:* `None`

Siehe auch: `orbit_framework.components.timer.ActivityTimerComponent`

O