Hochschule
Bonn-Rhein-Sieg
University
of Applied Sciences

Fachbereich Informatik
Department of Computer Sciences

Abschlussarbeit

Studiengang Bachelor Informatik

Embedding von MicroPython für das Open Robotics Board

von

Nils Hoffmann

Erstprüfer	Prof. Dr. Thomas Breuer
Zweitprüfer	Prof. Dr. Michael Rademacher
Betreuer	Beate Jost
eingereicht am	14.11.2024

Name:
Adresse:

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Unterschrift Sankt Augustin, den

Die schriftliche Ausarbeitung der Bachelorarbeit setzt sich gemäß Exposé aus den Entwicklungs-Dokumenten zusammen.
Alle Dokumente und die Arbeit selbst sind in einem GitHub-Projekt organisiert.
Diese Arbeit lässt sich unter: https://github.com/NiHoffmann/ORB/ abrufen.
Die Online-Version dieser Arbeit ermöglicht die Nutzung der Verlinkungen.

Anforderungen

Das Ziel dieses Projektes ist die Integration eines MicroPython-Interpreters in die Firmware des Open Robotic Board (ORB). Das ORB ist eine Hardwareplattform, die den modularen Aufbau eines Roboters ermöglicht. Die ORB-Firmware stellt Funktionen, wie z.B. die Motorsteuerung, für die Roboterprogrammierung bereit. Ziel ist es, die Entwicklung von Python-Programmen für das ORB zu ermöglichen. Zusätzlich soll die Anbindung für die Python-Programmierung an den ORB-Monitor umgesetzt werden. Der ORB-Monitor ist ein Entwicklungstool, das Textausgaben, das Hochladen, Starten sowie Stoppen von MicroPython-Programmen ermöglicht.

Für Debug- und Entwicklungs-Zwecke soll auch das Compilieren und Ausführen des Micropython-Ports auf einem Entwicklungs-Rechner bereitgestellt werden.

Es folgt eine genauere Liste und Beschreibung der daraus entstehenden Anforderungen:

1. Der Micropython Embed Port kann unter Windows kompiliert und debuggt werden.
2. Die Ausführung von Micropython-Skripten soll ausschließlich durch MPY-Bytecode ermöglicht werden, nicht durch Klartext oder eine REPL-Funktion.
3. Das Python-Programm soll im Flash-Speicher des ORB abgelegt werden.
4. Das Python-Programm soll vom Nutzer gestartet und gestoppt werden können.
5. Die Übertragung des Python-Programms soll über den von DFU bereitgestellten Prozess erfolgen können. Die Firmware soll nicht erneut aufgespielt werden müssen.
6. Das Python-Programms soll über die durch den ORB-Monitor bereit gestellten Prozesse übertragen werden können. Die Firmware soll nicht erneut aufgespielt werden müssen.
7. Es sollen Micropython-Module und -Funktionen bereitgestellt werden, die sich aus den in `orblocal.h` definierten Funktionen ableiten.
   Diese Funktionen werden von der ORB-Firmware für die Programmierung in C++ zur Verfügung gestellt:

```
virtual void        configMotor      (BYTE id,WORD t,BYTE a,BYTE Kp,BYTE Ki);
virtual void        setMotor         (BYTE id,BYTE m, short s, int p       );
virtual ORB::Motor  getMotor         (BYTE id                              );
virtual void        setModelServo    (BYTE id, BYTE s, BYTE w              );
virtual DWORD       getTime          (                                     );
virtual void        wait             (DWORD time                           );
virtual void        setMonitorText   (BYTE line,const char *fmt, va_list va);
virtual BYTE        getMonitorKey    (                                     );
virtual void        clearMemory      (                                     );
virtual void        setMemory        (DWORD addr, BYTE *data, DWORD s      );
virtual void        getMemory        (DWORD addr, BYTE *data, DWORD s      );
virtual void        configSensor     (BYTE id,BYTE t,BYTE m,WORD o         );
virtual ORB::Sensor getSensor        (BYTE id                              );
virtual WORD        getSensorValueExt(BYTE id, BYTE ch                     );
virtual BYTE        getSensorDigital (BYTE id                              );
```

[vgl. ORB-FW, 'ORB-Firmware/Src/ORBlocal.h', ab Zeile 47]

8. Die ORB-Funktionen müssen in ihrer vollständigen Funktionalität, in die MP-VM, integriert werden.
   Die Verwendung von Micropython darf keinen Funktionalitätsverlust zur Folge haben.
9. Diese Funktionen sollen nicht direkt übernommen, sondern in die Python-Welt übertragen werden. Falls angebracht werden Python-Objekte/Module bereitgestellt , welche die genannten Funktionen möglichst gut und mit geeigneten Name-Spaces abbilden.
10. Auch wenn die ORB-Funktionen vollständig umgesetzt werden sollen so reicht es für Funktionen welche meherere Komponenten geleichzeitig umsetzen, diese nur für ausgewählte Komponenten zu testen. Dies würde zum Beispiel die Sensor-Klasse betreffen. Es soll lediglich sicher gestellt werden, dass alle in Micropython eingestellten Parameter korrekt an die ORB-Firmware übergeben werden und einstellbar sind. Ist dies gegeben, so gilt eine Funktion als getestet.

11. Es sollen keine standardmäßigen Micropython-Module ge-ported werden. Es werden ausschließlich die Funktionen der ORB-Firmware bereitgestellt, sowie ausgewählte Hilfsfunktionen wie z.B. `math`, `min`, `max` usw..
12. Es soll eine klar definierte Schnittstelle zur Python-VM existieren, die als einzige Verbindung zwischen Micropython und der ORB-Firmware dient.
13. Es soll eine Python-Task implementiert werden, die parallel zu den anderen Aufgaben der ORB-Firmware läuft. Diese Python-Task nutzt die Schnittstelle zur Python-VM.
14. Die Funktionen der ORB-Firmware sollen lediglich erweitert und nicht grundlegend verändert werden.
    Die Python-Task muss im Einklang mit den restlichen Aufgaben der ORB-Firmware ausgeführt werden können.
    Anpassungen wie z.B. die Neubelegung von Tasten oder das Verschieben sowie Verringern von Nutzer-Speicherblöcken sind erlaubt, solange diese die Funktionalität der ORB-Firmware nicht einschränken.
15. Den Nutzern sollen ausreichende Informationen zur Verfügung gestellt werden, damit sie diese Funktionen verwenden können. Das heißt, auch ohne Vorkenntnisse über die ORB-Firmware soll es möglich sein, den Micropython-Interpreter zu nutzen. Dies sollte in der Form eines API-Dokumentes gestaltet werden.

Arbeitsplan

- [x] **1. Micropython Embed Port unter Windows**
  - [x] 1.1. Code::Blocks Umgebung einrichten
  - [x] 1.2. Kompilieren und Laden einer MPY-Binär-Datei
  - [x] 1.3. Erstes Modul registrieren & Mockup erstellen
  - [x] 1.4. Erster Entwurf der Micropython-VM Schnittstellen
  - [x] 1.5. User-Interrupt Integrieren & Thread-Safety Testen/Analysieren
  - [x] 1.6. Micropython- & Program-Compile-Prozess in den Code::Blocks Build Prozess einbauen

  > Nach diesem Schritt ist das Windows Grundgerüst gebaut.

  - [x] 1.7. Vollständiges Umsetzen der ORB-Funktions-Module & Mockups
  - [x] 1.8. Python-Api Dokumentation schreiben
  - [x] 1.9. Program-Compile-Prozess in Code::Blocks Build Prozess einbauen

  > Ab diesen Punkt sollten alle Funktionen unter Windows testbar sein.

  - [x] 1.10. Erstellen der Windows-Test-Spezifikation
  - [x] 1.11. Testen der Micropython-Funktionalitäten nach Spezifikation
- [x] **2. Integration in EMBitz Projekt**
  - [x] 2.1. Ausführen der VM auf einem Microcontroller (Mit MPY-Binary in die Firmware eingebaut - d.h. als uint_8t array).

  > Hier kann das ORB-Firmware Projekt als Grundgerüst genutzt werden, ohne das Verwenden der eigentlichen ORB-Funktionen. Funktionen wie usrLed.set(1) können erst einmal als Platzhalter in die ORB-C-Interfaces eingebaut werden.

  - [x] 2.2. Übertragen und Laden des Programmes aus dem Flash-Speicher

  > Verwenden des Entwicklungs-Gerüstes (2.1.).

  - [x] 2.3. Umsetzen der tatsächlichen C-Interface-Klassen
  - [x] 2.4. Erstellen der Python-Task
  - [x] 2.5. Entwicklungsgerüst erweitern um USB-Update

  > Für ORB-Monitor-Kommunikation.

  - [x] 2.6. Testen & Dokumentieren der ORB-Python-Funktionalitäten
  - [x] 2.7. Erster Entwurf der Micropython-VM-API Dokumentation

  > Festgehalten im Sphinx-Api-Doc.
- [x] **3. Vollständige Firmware Integration**
  - [x] 3.1. Wiederherstellung der ORB-Programm-Logik
  - [x] 3.2. Konflikte zwischen Tasks identifizieren & dokumentieren

  > Hier wäre ein Beispiel, dass das UserInterface Motoren ausschaltet, wenn keine AppTask läuft.

  - [x] 3.3. Konflike lösen & dokumentieren
  - [x] 3.4. Kontrollfluss Dokumentation (aus 3.2. & 3.3.) erstellen.

  > Welcher Button startet die Python-VM, welcher AppTask, etc. dieser Schritt ist abhängig von vorher getroffenen Designentscheidungen. Dies ist identisch zu der vorherigen Firmware, es musste nichts angepasst werden - daher fällt eine ausführliche Dokumentation weg.

  - [x] 3.5. Testen & Dokumentieren der ORB-Python-Funktionalitäten im Zusammenhang mit Kontrollfluss und ORB-Firmware Kompatibilität.
  - [x] 3.6. Micropython-VM-Api Dokumentation nachbessern

Implementierungsreport

Inhaltsverzeichnis

1. Einleitung

Der Implementierungsreport soll einen Überblick über die Schritte zur Umsetzung der ORB-Python-Firmware beschreiben. Also die Erweiterung der ORB-Firmware um die MicroPython-Byte-Code-Interpreter. Zunächst wird auf die Firmware-Architektur eingegangen. Dies soll die einzelnen Komponenten des Projektes kurz auflisten und ihre Aufgaben darstellen. Die Darstellung soll in Bezug zu dem vollständigen Projekt geschehen.

2. Firmware-Architektur

Im Folgenden wird kurz die Firmware-Architektur beschrieben. Hier anzumerken ist, dass wie in den Anforderungen beschrieben, auch unter Windows kompiliert und getestet werden soll. Die ORB-Firmware-System-Komponente wird unter Windows durch Platzhalter-Klassen, hier "Mockups" und Threads ersetzt. Durch diese Platzhalter werden die gleichen Schnittstellen wie durch die ORB-Firmware bereitgestellt.

Es gibt einige für diese Arbeit relevanten System-Komponenten. Im Wesentlichen lassen sich diese auf folgende Bestandteile eingrenzen. Die grün hinterlegten System-Komponenten sind Bestandteil der ursprüngelichen ORB-Firmware. Diese wurden gegebenenfalls erweitert oder angepasst, jedoch nicht gänzlich neu hinzugefügt. Die blau hinterlegten System-Komponenten sind solche, die durch das Anbinden der MicroPython-Virutal-Machine (MP-VM) angebunden und neu erstellt wurden.

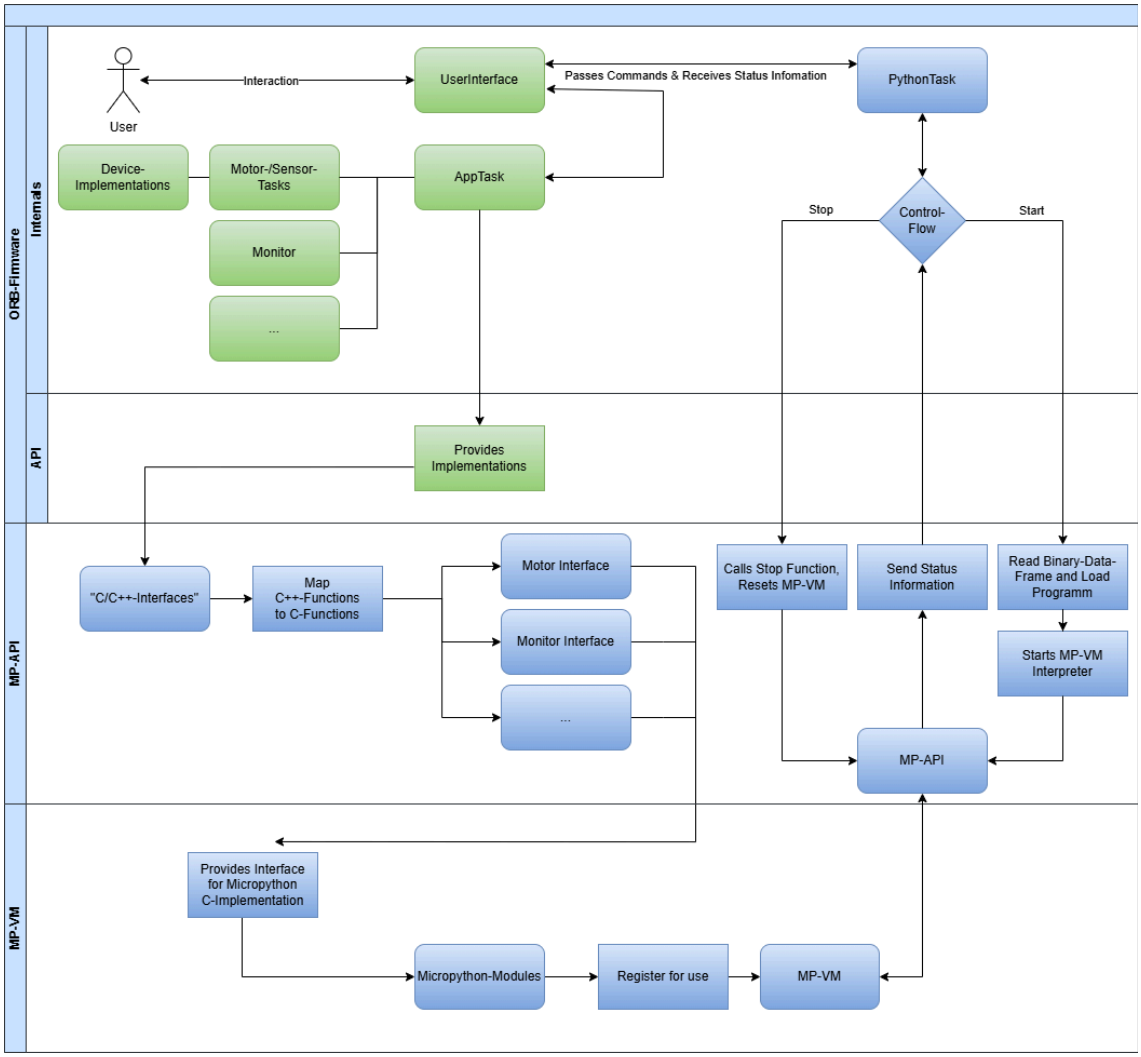

Abbildung 1: Firmware-Architektur

Die für dieses Projekt relevanten System-Komponenten lassen sich in 3 Hauptteile unterteilen. Diese sind die ORB-Firmware , MicroPython-API (MP-API) und die MicroPython-Virtual-Machine (MP-VM).

- ### 2.1 ORB-Firmware
  - #### 2.1.1. Internals

Die ORB-Firmware wird zum größten Teil bereits bereitgestellt. Sie enthält alle Funktionalitäten, die mit dem ORB umgesetzt werden sollen. Dazu gehört das Anbieten von RTOS-Tasks und Funktionen welche für die Programmierung des Roboters verwendet werden. Außerdem gibt es ein User-Interface, so wie Bluetooth und USB-Schnittstellen. Desweiteren wird durch die ORB-Firmware eine Verbindung zu dem ORB-Monitor bereitstellt. Der ORB-Monitor ist ein Entwicklungs-Tool. Dieses wird für die Programmübertragung und zur Kommunikation mit dem Roboter verwendet. Die MicroPython-VM soll die hier genannten Funktionalitäten verwenden und in diese integriert werden.

2.1.2. API

Für die Programmierung unter C++ stellt die ORB-Firmware bereits eine API-Klasse bereit, die `ORB-Local.h`. Die MicroPython-VM soll, wie in den Anforderungen beschrieben, diese Funktionen auch zur Verfügung stellen. Für diesen Zweck wird die die AppTask als Interface-Klasse verwendet. Diese bietet die gleichen Funktionen wie die `orb_local.h` an. Die umgesetzen Module der MicroPython-VM rufen im wesentlichen die dort definierten Funktionen auf.

2.1.3. Python-Task

Der MicroPython-Interpreter soll in sich geschlossen als Teil der ORB-Firmware ausgeführt werden. Dazu soll eine MicroPython-RTOS-Task implementiert werden, im Folgenden auch Python-Task genannt. Diese verwendet durch die MP-API bereitgestellte Funktionen. Die Python-Task soll die VM starten, stoppen und diese allgemein verwalten. Zu diesem Zweck werden auch Funktionen zur Status-Abfrage der MP-VM implementiert. Die Python-Task wird mit dem User-Interface verbunden. Diese soll genauso wie die App-Task verwendet werden können.

2.2. MicroPython-API (MP-API)

2.2.1. C/C++ Interfaces

Da die MP-VM ein C-Projekt ist und die ORB-Firmware C++-Funktionen anbietet, ist es wichtig, das C++-nach C-Interfaces bereitgestellt werden. Die Interfaces werden in den meisten Fällen nur einen Aufruf der C++-Funktion innerhalb einer C-Funktion darstellen. In manchen fällen müssen umfangreichere änderungen vorgenommen werden.

2.2.2. MicroPython-API

Die MP-API ist die Schnittstelle zu der MP-VM, hier werden Funktionen für die Python-Task bereitgestellt. Diese sollen das Einbinden in die ORB-Firmware erleichtern und klar strukturieren. Wie bereits in 2.1.3. Python-Task beschrieben, werden hier Funktionen zur Verwaltung der Python-Task bereitgestellt.

2.3. MicroPython-Vitrual-Machine (MP-VM)

Die MicroPython-Vitrual-Machine (MP-VM) ist der MicroPython-Byte-Code-Interpreter. Dieser führt MicroPython-Byte-Code aus. Alle durch MicroPython verwendbaren Module müssen für diese definiert und registriert werden.

2.3.1. MicroPython-Modules

Im Wesentlichen werden hier die MicroPython-Module umgesetzt. Diese definieren mit welchen Objekten, Funktionen, etc. das ORB programmiert werden kann. In diesem Schritt werden die C-Interface-Klassen an die MP-VM angebunden. Diese Module sind Abbildungen der ORB-Funktionen und keine genaue Kopie. Funktional sollen sie jedoch die gleichen Konzepte abbilden.

2.3.2. MP-VM

Die MP-VM wird durch das MicroPython-Projekt bereitgestellt. Die MP-VM wird für den Anwendungsfall dieses Projektes konfiguriert, angepasst und durch Funktionen, welche für die Umsetzung dieses Projektes benötigt werden, erweitert.

Wie die genauen Anforderungen an die umzusetzenden Funktionen aussehen, sowie in welchem Maße die ORB-Firmware durch die MP-VM erweitert werden soll, ist dem 'Anforderungen'-Dokument zu entnehmen. Auf welche Art und Weise diese Ziele erfüllt wurden, wird im Folgenden beschrieben.

3. Aufsetzen des GitHub-Projekts

1. **Repository erstellen**:
    - Zunächst musste ein neues Repository auf GitHub angelegt werden. Es wurde eine README.md erstellt. Diese ReadeMe-Datei soll Anweisungen zu der Verwendug bzw. Einrichtung dieses Projektes enthalten und im Laufe des Projektes erweitert werden.
2. **Fork des MicroPython-Projekts erstellen**:
    - Der erste Schritt hier war es einen Fork des MicroPython-Projektes zu erstellen.
      Durch diesen Schritt ist es einfacher nachzuvollziehen, an welchen Stellen Änderungen in dem MicroPython-Projekt vorgenommen wurden.

> Note
>
> Der Embed-Port des Projektes wird gegen den Embed-Port im MicroPython-Projekt gebaut, d.h. Änderungen an dem Code von MicroPython wie z.b. Änderungen an der `vm.c` (wie z.B. 6.2. MicroPython-Ausführung unterbrechen) werden in diesem Fork gemacht.

3. **MicroPython als Submodul hinzufügen**:
    - Dadurch das ein Fork des MicroPython-Projektes erstellt wurde, konnte dieses als Submodul in das GitHub-Projekt integriert werden. Dazu wurde folgender Befehl ausgeführt:

      ```
      git submodule add https://github.com/NiHoffmann/micropython micropython
      ```

      Durch die Submodul-Struktur ergibt sich eine klare Trennung der verschiedenen Komponenten dieses Projektes. Auch alle anderen Git-Repositories die im Laufe des Projektes eingebunden wurden, werden als Submodul bereitgestellt. Dies soll auch späteren Entwicklern die Möglichkeit geben, dieses Projekt bei Änderungen, Bug-Fixes, usw. diese möglichst einfach wieder in das ORB-Python Projekt einzubinden.
4. **MicroPython-Projekt einrichten**:
    - Anschließend musste der MPY-Cross-Compiler gebaut werden. Dieser wird verwendet um Python-Programme zu MPY-Byte-Code zu kompilieren. Der MPY-Byte-Code ist das Daten-Format, welches von der MicroPython-VM gelesen und verarbeitet werden kann. Dafür wird in dem Verzeichnis micropython\mpy-cross der Befehlt `make` ausgeführt.
    - Im Gegensatz zu anderen MicroPython-Ports hat der Embed-Port keine externen Abhängigkeiten. Daher musste dieser nicht weiter eingerichtet werden.
5. **Erste Projektkonfiguration mit MicroPython Embed**: Für die Initialisierung des Projekts wurden erst einmal zwei Dateien aus dem MicroPython-Embed-Projekt verwendet:
    - micropython_embed.mk: Diese Datei enthält die Make-Regeln für den Embed-Port.
    - mpconfigport.h: Diese Datei enthält die Konfiguration der MicroPython-Ports mehr dazu, in Konzepte, unter MicroPython-Flags.
    - Diese beiden Dateien werden in dem Libs-Ordner unter ORB-Python\libs\mp_embed abgelegt. Sie werden verwendet um dem MicroPython-Embed-Port zu bauen.
6. **Änderungen am geforkten MicroPython-Projekt**:
    - Die mpconfigport_common.h-Datei ist für den Build-Prozess vorgesehen und enthält Definitionen für z.B. die Speicherzuweisung. Unter Linux wird hier der Header `alloca.h` eingebunden. Da der äquivalente Windows-Header `malloc.h` fehlt, wurde eine Elif-Anweisung hinzugefügt.

Nach: [MP, micropython/ports/embed/port/mpconfigport_common.h], ab Zeile: 36:

```
#if defined(__FreeBSD__) || defined(__NetBSD__)
#include <stdlib.h>
#elif defined(_WIN32)
#include <malloc.h>
#else
#include <alloca.h>
#endif
```

Dies ermöglicht das Bauen des MicroPython-Port unter Windows.

7. **Pfadanpassung in micropython_embed.mk**:
   - In der Make-Datei micropython_embed.mk habe der Pfad für `MICROPYTHON_TOP` zu dem Relativen-Pfad meines MicroPython-Projektes angepasst:

     ```
     MICROPYTHON_TOP = ../../../micropython
     ```

8. **Kompilieren des MicroPython-Projekts**:
   - Anschließend konnte das MicroPython-Projekt vor-kompiliert werden. Dafür musste sichergestellt werden, dass alle notwendigen Werkzeuge wie Make, GCC und Python installiert sind. Die Kompilierung wurde mit folgendem Befehl gestartet:

     ```
     make -f .\micropython_embed.mk
     ```

9. **Einrichten des Code::Blocks-Projekts**:
   - Anschließend wurde ein leeres C++-Projekt in Code::Blocks erstellt. Dieses wurde so konfiguriert, dass der GCC-Compiler verwendet wird.

10. **main.cpp erstellen und Suchverzeichnisse konfigurieren**:
    - Um die Code::Blocks-Konfiguration zu vallidieren, wurde eine main.cpp-Datei erstellt. Zusätzlich wurden die Suchverzeichnisse für den Compiler angepasst. Folgende Pfade mussten hinzugeführt wuerden:
      - `src/`
      - `libs/micropython_embed`
      - `libs/micropython_embed/port`
    - Zusätzliche wurden die benötigten C-Dateien des MicroPython-Port zu dem Code::Blocks-Build-Prozess hinzugefügt. Dies sind alle von MicroPython im Vor-Kompilier-Prozess generierten Dateien. Weitere im Verlauf erstellte Dateien und Verzeichnisse müssen auch zu den Build-Dateien und Code::Blocks-Build-Prozess hinzugefügt werden.

11. **Compiler-Flags hinzufügen**: Um sicherzustellen, das MicroPython korrekt kompiliert und problemlos ausgeführt werden kann, ist es wichtig die richtigen Compiler-Flags zu setzen. Das MicroPython-Projekt stellt eine Vielzahl an Mikrocontroller-Ports bereit. Diese wurden betrachtet. Es konnte folgende Erkenntnis daraus gezogen werden. Die '-Og' und '-Os' Flags werden für MicroPython-Ports verwendet. Ansonsten wird das Kompilieren mit '-O0' als unproblematisch angesehen. Die '-O0'-Flag wird für die Optimierungsfreie-Kompilierung verwendet. Jedoch ist hier ein unter Windows auftretender Bug zu beachten, siehe Windows-Bug: Falsches Register bei Non-Local Return-Adressierung. An dieser Stelle sind Kompatibilitätsüberlegungen der Compiler-Flags, unter Berücksichtigung der ORB-Firmware, zu machen. Im Konzept unter Compiler Flag Kompatibilität ist genauer erklärt, warum es als notwendig angesehen wird, nur diese Flags zu verwenden, falls Optimierung gewünscht oder notwendig ist.

12. **Code::Blocks Build Targets anpassen**: Zu diesem Zeitpunkt war der Build-Prozess in der Code::Blocks-Umgebung von dem des MP-Embed-Port losgelöst. Im besten Fall sollte aus der Code::Blocks-Umgebung auch der MP-Embed-Port gebaut werden können. Zu diesem Zweck wurden im Code::Blocks-Projekt 2 Build-Target angelegt :*Build*und*Rebuild*. Während Build der vorher konfigurierte Prozess ist, ist Rebuild erweitert durch einen Pre-Build-Step. Das*Rebuild*-Build-Target ruft ein Batch-Script auf welches die Kommandos für das 'Clean' und 'Build' des MicroPython-Embed-ports enthält. Die Verwendung des Targets ist wie folgt gedacht. Entwickelt man nur in der Code::Blocks-Umgebung ohne neue Module oder Funktionen hinzuzufügen, so reicht das Build-Target aus, da die MP-Embed-Port-Ressourcen nicht immer neu generiert werden müssen. Kommen nun neue Module hinzu, werden alte umbenannt oder neue Funktionen eingeführt. So kann man einmal das Rebuild-Target ausführen. Danach verwendet man wieder wie gewohnt Build.

4. MicroPython Modul Registrierung

4.1. Registrieren eines Test-Moduls

Als erstes Modul wurde ein Test-Modul umgesetzt. Dies war eine direkte Kopie des examplemodule.c , welche nicht weiter konfiguriert wurde.

Um dieses mit dem Embed-Port dieses Projektes verwenden zu können, wurde in dem `src` -Verzeichnis ein Modul-Ordner erstellt. Es musste in dem MicroPython-Embed-Makefile (ORB-Python/libs/mp_embed/micropython_embed.mk) die `USER_C_MODULES` -Variable um den Pfad zu den Modul-Ordnern erweitert werden. Außerdem wurde in diesem Schritt direkt die C-Flag um die Includ-Directive ‚-I' für den Modul-Ordern erweitert. Die Include-Pfade für Module sollen immer relativ zu diesem Ordner gesetzt sein, um die `USER_C_MODULES` nicht für jedes Modul erneut anpassen zu müssen.

Das durch MicroPython bereit gestellte Beispiel-Modul konnte problemlos eingebunden werden, wodurch validiert werden konnte, dass das Projekt an dieser Stelle korrekt konfiguriert wurde.

Zu diesem Zeitpunkt wurde jedoch noch der Klar-Text-Interpreter des MicroPython-Projektes verwendet.

4.2. Compilieren und Ausführen einer MPY-Binär-Datei

Im nächsten Schritt wurde der Byte-Code Interpreter getestet. Dafür ist es wichtig das folgende Flags in der mpconfigport.h gesetzt sind:

```
#define MICROPY_PERSISTENT_CODE_LOAD            (1)
#define MICROPY_ENABLE_GC                       (1)
#define MICROPY_PY_GC                           (1)
```

Wobei `MICROPY_PERSISTENT_CODE_LOAD` die Flag für das Verwenden des Byte-Code-Interpreters ist. Die anderen beiden Flags werden für den Garbage-Collector verwendet. Zu diesem hat der MicroPython-Embed-Port Abhängigkeiten und diese müssen für diesen Port immer eingestellt sein. Nachdem dies konfiguriert war, konnte das Code::Blocks-Projekt erweitert werden. Zunächst wurde hier eine MPY-Datei in einem Byte-Array statisch gespeichert. Diese konnte dann an die MP-VM gegeben und ausgeführt werden.

Durch diese Umsetzung war es möglich einen ersten Funktions-Test des Byte-Code-Interpreter durchzuführen. Nach dieser validiert werden konnte, war der nächste Schritt das Laden einer MPY-Datei durch das Windows-File-System. Nachdem auch dies umgesetzt war, war es möglich mit dem Aufruf 'mpy-cross program.py' MicroPython-Programe zu compilieren und die MPY-Datei von dem Code::Blocks-Projekt laden zulassen.

Die aus der Code::Blocks-Umgebung geladene Datei wird in dem Pfad `ORB-Python\program\program.mpy` erwartet. Dies hat sich bis zu dem Ende dieses Projektes nicht verändert. Jedoch wurde das Datei-Format der erwarteten Binär-Datei angepasst und auch erweitert. So wird bei dem aktuellen Stand eine `.bin` -Datei erwartet. Außerdem ist es möglich mithilfe verschiedener Start-Parametern MicroPython-Dateien aus anderen Ordnern auszuführen. Ein Beispiel wäre das Ausführen der Windows-Tests. Für diesen Zweck werden zwei Parameter unterstützt. Diese Parameter sind die Übergabe des Pfades zu der MPY-Datei und das Setzen einer Flag, welche bestimmt ob der MicroPython-Interpreter durch einen User-Interrupt unterbrochen werden soll.

```
ORB-Python.exe "path_to_bin/file.bin" "execute_with_interrupt"
```

Die genannte .bin Datei wird durch ein zusätzlich erstelltes Compile-Script realisiert. Die Implementierung des Compile-Scripts ist in zwei Kapitel unterteilt: 7. Compile-Script erster Entwurf und 10. User-Program-Compile-Script vollständig umgesetzt. Die gesamten Funktionalitäten des Compile-Prozesses wurden im Laufe des Projektes auf das Tool 'orb-util' verlagert. Dieses bündelt die verschiedenen erstellen Scripts, welche für die Verwendung des ORB benötigt werden. Eben so wie solche, die für eine benutzerfreundliche Verwendung entstanden sind.

4.3. ORB-Python-Module hinzufügen

4.3.1. Klassen, Objekt und Funktionen ermitteln

Der erste Schritt für die Umsetzung der ORB-Python-Module, war es zu überlegen welche Module umgesetzt werden sollen. Wie den Anforderungen zu entnehmen ist, sollen alle Funktionen der orblocal.h in MicroPython abgebildet werden. Dazu gehört es nicht nur die Funktionen abzubilden, sondern auch zu überlegen, wie die Funktionen am besten in die "Python-Welt" übertragen werden können.

Hier ist es sinnvoll Funktionen in Klassen zu bündeln. Z.b. Motor-Funktionen in eine Motor-Klasse. Geräte, welche an das ORB angeschlossen werden, können in einem Modul 'devices' zusammengefasst werden. Alle anderen Funktionen könnten in einem gesonderten Modul wie z.b. die 'wait()'-Funktion eines 'time'-Moduls gebündelt werden.

Die genaue umgesetzte Strukturierung ist der Python-Api: "sphinx-python-api.pdf" zu entnehmen. Ursprünglich war der Plan mit zusätzlichen Name-Spaces zu arbeiten bzw. allen Modulen einen 'orb.' Präfix zu geben. Dies würde verdeutlichen, dass es sich hier um die ORB-Implementationen handelt. Davon musste jedoch absehen werden: siehe Problematik bei der Verwendung von Namespaces.

4.3.2 Aufbauen der Module

In diesem Schritt wurden die Module, Klassen, Objekte und Funktionen des MicroPython-Port erstellen. Abgeleitet werden diese wie bereits beschrieben aus der orblocal.h . Der Modul-Design-Flow für MicroPython-Module hat sich als ein klarer Prozess herausgestellt. Dieses ist in dem folgenden Diagramm beschrieben.

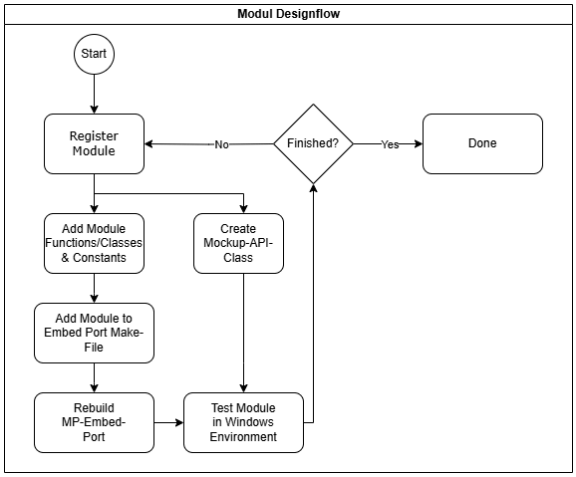

Abbildung 2: Module Designflow

Im Wesentlichen mussten hier Dictionaries für Module, Objekte, Konstanten und Funktionen anlegt werden. Diese werden verwendet, um C-Funktionen an einen MicroPython-Aufruf zu binden. Die Registrierung dieser Module verwendet für die Namen der Aufrufe im Python-Code QStrings. Durch die Natur der QStrings, musste nach dem Erstellen oder Umbenennen von Modulen oder Funktionen, der MicroPython-Embed-Port neu gebaut werden. Mehr dazu im Konzept unter QString. Im Wesentlichen wird hier das C-Bindeglied zu dem MicroPython-Interpreter erstellt. Auch die MicroPython-C-Interface-Klassen bzw. deren Mockup-Varianten wurden in diesem Schritt erstellt. Mehr dazu und der Zweck der Mockup-Klassen im nächsten Kapitel 4.3.3. Code::Blocks und Mockups. Im Folgenden ein übersichtliches Beispiel-Modul, welches das Vorgehen bei der Entwicklung eines MicroPython-Moduls deutlich machen soll:

```c
//Include für die MicroPython Methoden und Objekt-Definitionen
#include "py/builtin.h"
#include "py/runtime.h"

//Erstellen einer C-Funktion, diese wird von dem MicroPython Interpreter Später
aufgerufen
static mp_obj_t py_subsystem_info(void) {
    //Der Rückgabe wert von MicroPython Funktionen ist immer ein MicroPython-Objekt
    return MP_OBJ_NEW_SMALL_INT(42);
}
//Mocro um eine Funktion als MicroPython-Objekt zu registrieren
MP_DEFINE_CONST_FUN_OBJ_0(subsystem_info_obj, py_subsystem_info);

//Modul Dictionary
static const mp_rom_map_elem_t mp_module_subsystem_globals_table[] = {
    //Name des Moduls
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_subsystem) },
    //QString mit Wert `info` welcher an den Funktions-Aufruf subsystem_info_obj
gebunden wird.
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&subsystem_info_obj) },
};
//Macro um das Dictionary zu einem MicroPython-Objekt zu konvertieren
static MP_DEFINE_CONST_DICT(mp_module_subsystem_globals,
mp_module_subsystem_globals_table);

//Das hier ist unser MicroPython-Objekt für das Modul
const mp_obj_module_t mp_module_subsystem = {
    //Wir haben einen Objekt-Typ-Modul
    .base = { &mp_type_module },
    //Hier definieren wir den gloabls-table also die öffentlich verfügbaren Funktionen
dieses Moduls
    .globals = (mp_obj_dict_t *)&mp_module_subsystem_globals,
};

//Tatsächliche Registrierung des Moduls, dieser Aufruf fügt das Modul in die
MicroPython-Modul/Funktions "Tabelle" ein
MP_REGISTER_MODULE(MP_QSTR_subsystem, mp_module_subsystem);
```

[vgl. MPD, „Implementing a core module“]

Wie genau Module aufgebaut werden und die Registrierung aussieht, ist aus dem Konzept zu entnehmen. Unter dem Kapitel MicroPython-Types findet sich ein guter Einstiegspunkt. Es finden sich noch weitere Kapitel, welche das interne Vorgehen des MicroPython-Interpreters erläutern. Hier wird auch genauer auf die Bausteine der Implementierung eingegangen. Diese sind weitestgehend innerhalb des genannten Kapitels verlinkt.

4.3.3. Code::Blocks und Mockups

Wie in Warum den MicroPython Embed Port verwenden? angedeutet, soll der MicroPython Embed Port ohne die direkte Einbindung der ORB-Firmware kompiliert werden können. Zu diesem Zweck werden sogenannte "Mockups" eingeführt. Diese Mockups sind C-Dateien, welche die C-Interface-Methoden der ORB-Firmware simulieren. Die tatsächlichen Implementierungen werden später als Teil des ORB-Firmware-Projektes umgesetzt.

Das bedeutet, dass Mockups die gleichen Konstanten, Funktionen und Variablen enthalten, welche später von der ORB-Firmware benötigt werden. Diese stellen die Funktionalitäten des ORB für die MicroPython-Umgebung bereit. Ein Beispiel könnte folgendermaßen aussehen:

Es gibt eine Motor-Mockup-Datei mit der folgenden Funktion:

```
void setMotor(uint8_t port, uint8_t mode, int16_t speed, int pos) {
    printf("set motor port(%u) mode(%u) speed(%d) pos(%d)\n", port, mode, speed, pos);
}
```

Das Motor-Modul kann dann diese Mockup-Funktion verwenden:

```
static mp_obj_t set(size_t n_args, const mp_obj_t *pos_args, mp_map_t *kw_args) {
    enum { ARG_mode, ARG_speed, ARG_position };

    static const mp_arg_t allowed_args[] = {
        { MP_QSTR_mode, MP_ARG_REQUIRED | MP_ARG_KW_ONLY | MP_ARG_INT, {.u_int = 0 } },
        { MP_QSTR_speed, MP_ARG_KW_ONLY | MP_ARG_INT, {.u_int = 0 } },
        { MP_QSTR_position, MP_ARG_KW_ONLY | MP_ARG_INT, {.u_int = 0 } },
    };

    mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
    mp_arg_parse_all(n_args - 1, pos_args + 1, kw_args, MP_ARRAY_SIZE(allowed_args),
allowed_args, args);

    motor_obj_t *self = MP_OBJ_TO_PTR(pos_args[0]);

    //use the mockup funktion
    setMotor(self->port, args[ARG_mode].u_int, args[ARG_speed].u_int * self->direction,
args[ARG_position].u_int * self->direction);


    return MP_OBJ_FROM_PTR(self);
}
static MP_DEFINE_CONST_FUN_OBJ_KW(set_obj, 1, set);
```

Im Gegensatz dazu wird die ORB-Firmware anstelle der Mockup-Datei eine andere Motor-Implementations-Datei einbinden. Für beide Dateien sind Methoden, Namen und Parameter gleich. Sie bieten somit ein C-Interface für die C++ Funktionen der ORB-Firmware. Dieser Schritt wird über einen projektspezifischen Include-Ordner für die Interface-Dateien realisiert. Die Struktur innerhalb der Includ-Ordner ist gleich.

5. Erstellen der Sphinx-Dokumentation

Im nächsten Schritt wurde die Sphinx-Dokumentation erstellt. Diese enthält Informationen über die Python-API, also welche Funktionen aus der MP-VM erreichbar sind.

5.1. Aufsetzen der Sphinx-Dokumentation

Um die Sphinx-Dokumentation einzurichten, wurde den Anweisungen aus https://www.sphinx-doc.org/en/master/usage/installation.html gefolgt. Alle weiteren relevanten Informationen lassen sich ebenfalls in der Sphinx-Dokumentation finden. Dort findet sich auch die Syntax zur Erstellung der Dokumentation. Da Sphinx nicht direkt Gegenstand dieser Bachelorarbeit ist, wird auf diese nicht weiter eingegangen. Die Sphinx-Tools wurden mithilfe von Chocolatey installiert. Als Grundgerüst der Dokumentation wurde mit `sphinx-quickstart docs` ein Dokumentations-Layout generiert. Durch Sphinx ist es möglich, die Dokumentation als HTML-Seite anzubieten. Diese kann unter: `Dokumentation/sphinx-docs/build/html/index.html` aufgerufen werden. Dadurch das Sphinx es auch erlaubt die Dokumentation zu Latex zu konvertieren, wird auch eine Dokumentations-PDF zur Verfügung gestellt. Mithilfe der Datei `Dokumentation\sphinx-docs\make.bat` können sowohl die HTML-, als auch die Latex-Dokumentationen generiert werden.

```
make.bat html
make.bat latex
```

5.2. Umfang der Dokumentation

Die vollständige Dokumentation umfasst alle Funktionen die in der MP-VM umgesetzt wurden. Es sind einfache Informations-Texte zu den einzelnen Modulen verfasst worden. Es wurde, wie in den Anforderungen definiert, die Python-API dokumentiert.

Abbildung 3: Python-API

Ebenso wurden später die Python-VM-Schnittstellen-Funktionen dokumentiert. Die Python-VM-Schnittstellen-Funktionen wurden im weiterer Verlauf dieses Projektes umgesetzt. Im nächsten Kapitel wird auf diese genauer eingegangen. Es sind solche Funktionen, welche von der ORB-Firmware verwendet werden, um die MP-VM zu programmieren.

6. Entwurf der MicroPython VM-Schnittstelle

Die öffentliche Schnittstelle, der Python-VM, sieht wie folgt aus:

Aus: 'ORB-Python/src/python-vm/python-vm.h':

```
void run(LoadLengthFunction loadLength, LoadProgramFunction loadProgram);
bool isRunning();
void stopProgram();
int getExitStatus();
const char* getExitInfo();
```

Wie zu erkennen ist, werden die Start- und Stop-Funktionen umgesetzt. Zusätzlich gibt es, den Status der Ausführung, 'isRunning()'. Diese Funktionen sind die wichtigsten Funktionen der Schnittstelle und umfassen ihre grundlegenden Aufgaben. Diese werden von der ORB-Firmware zur Verwaltung des MicroPython-Interpreters verwendet. Hier zu erkennen ist, dass es zusätzlich Funktionen für die Exit-Info und den Exit-Status gibt. Der Exit Status ist der aktuelle Exit-Status-Code der VM. Also ein Wert aus einem Enum, welcher für eine bestimmte Art der VM-Beendung steht. Die Werte des Exit-Status werden in der Config-Port-Header-Datei definiert. Die Exit-Info ist eine Zeichenkette, welche weitere Informationen über die Art der VM-Beendung enthält. Führt das Ausführen der MP-VM zu einer Exception, so wird hier die Art der Exception ausgegeben. Bei einem Import, welcher nicht aufgelöst werden konnte, ist der zurückgegebene Wert 'Import Error' und der zurückgegebene Exit-Status "1". Während der Exit-Status für alle Exceptions gleich ist, kann die Exit-Info variieren.

Vgl.: 'ORB-Python/src/python-vm/orb_config_port.h':

```
//Rückgabewert einer normalen Ausführung, Programm ist zu Ende.
#define ORB_EXIT_NORMAL                         (0)
//Es wurde eine Exception geworfen und nicht gehandelt.
#define ORB_EXIT_EXCEPTION                      (1)
//Es wurder der User-Interrupt (z.B. Stop-Button) verwendet.
#define ORB_EXIT_INTERRUPT                      (2)
```

Eine weitere Besonderheit ist die Art und Weise, wie die 'run' Funktion aufgerufen wird, da dieses Interface sowohl von der Windows-Umgebung, als auch Firmware-Umgebung verwendet wird. Hier musste das Umsetzen des Programm-Byte-Code-Ladens von der VM-Schnittstelle gelöst werden. Dazu wurden die folgenden Funktionen als Parameter für die 'run'-Funktion definiert:

Aus: 'ORB-Python/src/python-vm/python-vm.h':

```
typedef uint8_t* LoadProgramFunction(int length);
typedef uint32_t LoadLengthFunction();
```

Im Falle der Code::Blocks-Umgebung sind diese Funktionen in der main.cpp definiert. Sie laden den Programm-Code aus einer durch ein Char-Array definierten Datei. Möchte man die Code::Blocks-Umgebung zum Testen verwenden muss dieser Pfad angepasst werden.

Aus: 'ORB-Python/src/main.cpp':

```
char name[] = "program\\program.bin";
```

Die ORB-Firmware soll diese Funktionalitäten in der Python-Task umsetzen. Die Python-Task verwendet dabei Zugriffe auf den Flash-Speicher, um das Programm korrekt zu laden. Hier zu beachten ist, dass die VM-Schnittstelle in beiden Fällen ein durch `malloc` erzeugtes Byte-Array als Rückgabewert der Loadprogramm-Funktion erwartet. Die `free` -Funktion wird intern auf dem Byte-Array, durch die 'run'-Funktion, aufgerufen.

6.1. Ausführung der VM in einem Thread

Um die oben genannten Funktionen testen zu können, war es notwendig, die MP-VM parallel zu anderer Programmlogik ausführen zu können. Für diesen Zweck wurde in der Code::Blocks-Umgebung das Ausführen der MicroPython-VM in einem separaten Thread gestartet. Dadurch konnten die MP-VM-Schnittstellen-Funktionen genauer untersucht werden, wie z.B. für das Stoppen der Programmausführung. Wie genau dies realisiert ist, wird in dem folgenden Kapitel 6.2. MicroPython-Ausführung unterbrechen beschrieben.

6.2. MicroPython-Ausführung unterbrechen

Für das Anbinden des Python-VM muss es eine Möglichkeit geben, die VM-Ausführung durch die ORB-Firmware zu unterbrechen. Gerade für den Fall, dass ein Nutzer-Programm in einer Endlos-Schleife hängen bleibt. Betrachten wir bereits umgesetzte Systeme des MicroPython-Projektes, wie z.B. den Microbit-MicroPython-Port, können wir folgende Erkenntnis ziehen: Die meisten bereits bestehenden Systeme setzen einfach den gesamten Mikrocontroller zurück, um die Programm-Ausführung zu stoppen. Dies ist für einen Mikrocontroller der nur MicroPython ausführt eine gute Lösung. Für diesen Anwendungsfall jedoch eher unpassend.

Microbit:

```
static mp_obj_t microbit_reset_(void) {
    NVIC_SystemReset();
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_0(microbit_reset_obj, microbit_reset_);
```

[MP, 'micropython/ports/nrf/boards/MICROBIT/modules/modmicrobit.c', ab Zeile: 38]

oder auch Zephyr:

```
static mp_obj_t machine_reset(void) {
    sys_reboot(SYS_REBOOT_COLD);
    // Won't get here, Zephyr has infiniloop on its side
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_0(machine_reset_obj, machine_reset);
```

[MP, 'micropython/ports/zephyr/modmachine.c', ab Zeile: 48]

Da die MP-VM für diese MicroPython-Implementation durch einen separaten Task realisiert wird, sollte nicht der gesamte Microcontroller bei einem Programm-Stop zurückgesetzt werden. Hier muss eine alternative Lösung gefunden werden.

Diese Beobachtung wurde durch das Betrachten von MicroPython-Community-Chats bestätigt: „You can reset the processor from within an ISR using pyd.hard_reset() or machine.reset(). Otherwise, you'd need to set a flag and have the main script exit when it detects that flag is set.“ [MPC]

Die vorgeschlagene Lösung hier besteht darin, ein neues Flag einzuführen. Dies ist mit den in Thread Safety beschriebenen bedenken zu vereinbaren. Die Überprüfung dieser Flag wurde in die Datei 'vm.c', am Anfang der 'dispatch_loop' aufgenommen. Diese Schleife ist die Logik, welche bestimmt, wie eine Byte-Code-Anweisung verarbeitet werden soll.

Dabei sollte sich der MicroPython-Interpreter genauso wie bei einer vom Programm-Code erzeugten Exception verhalten, welche nicht durch einen 'catch'-Block abgefangen wird. Ziel ist, dass die MicroPython-VM in einem gültigen bzw. ihr bekannten Zustand beendet wird. Dadurch wird gewährleistet, dass die MicroPython-VM auch nach dem Stoppen problemlos neu gestartet werden kann.

Die Implementierung dieses Prozesses sieht wie folgt aus: Von außerhalb der VM muss im Falle eines Interrupts ein Flag, hier 'orb_interrupt', gesetzt werden:

Nach: [MP, 'micropython/py/vm.c', ab Zeile: 309]:

```
<...>
dispatch_loop:
       //This is the Main Logic, orb_interrupt will only ever be written from outside mp
       //so this flag is never a race condition, although we might finish one more
"cycle" of micropython
       //execution but that is fine.
       //inside here we create the exception so we never get mem error
       //we have to do this at the top to bypass controll flow
       #ifdef ORB_ENABLE_INTERRUPT
       //Überprüfen ob wir einen Interrupt haben
       if(MP_STATE_VM(orb_interrupt)){
           //custom exception Objekt anlegen
           static mp_obj_exception_t system_exit;
           system_exit.base.type = &mp_type_SystemExit;
           //da es sich um einen User-Interrupt handelt bleibt der Stack-Trace leer
           system_exit.traceback_alloc = 0;
           system_exit.traceback_data = NULL;

           //Wir übergeben nur ein Argument zusammen mit unserer Exception, den Namen
der Exception "User Interrupt"
           system_exit.args = (mp_obj_tuple_t*) mp_obj_new_tuple(1, NULL);
           mp_obj_t mp_str = mp_obj_new_str("User Interrupt", 14);
           system_exit.args->items[0] = mp_str;
           //wir setzen die MicroPython-State auf unsere erstellte Exception
           MP_STATE_THREAD(mp_pending_exception) = &system_exit;
           //Diese zweite Flag soll Race-Condiditons vermeiden, da es einen zweiten Teil
der User-Interrupt-Logik gibt.
           MP_STATE_VM(orb_interrupt_injected) = true;
       }
       #endif

<...>
```

Es ist wichtig zu beachten, dass dieses Flag nicht von innerhalb des MicroPython-Projekts beschrieben wird, auch nicht zum Zurücksetzen des Flags.

Dieser Code plant eine einfache Exception. Der Nachteil hier ist, das dies mit einem unveränderten MicroPython von einem Try-Catch-Block abgefangen werden kann.

Somit musste hier noch eine weitere Änderung an der MP-VM vorgenommen werden. Im Folgenden die Änderung welche Try-Catch-Logik, falls das Interrupt-Flag gesetzt, ignoriert:

Nach: [MP, 'micropython/py/vm.c', ab Zeile: 1473]:

```
<...>
//Diese Abfrage prüft, wie die Tiefe der Exception sich zu der Tiefe der gehandelten
Exceptions verhält.
if (exc_sp >= exc_stack
```

```
    //this part of the code handles try/catch blocks, we dont want them to be treated as
such, if orb interrupts treat any error as un-handled
    #ifdef ORB_ENABLE_INTERRUPT
    && !MP_STATE_VM(orb_interrupt_injected)
    #endif
    ) {
    // catch exception and pass to byte code
) {
<...>
```

Nun kann der VM-Interrupt geplant werden. Da die Dispatch-Schleife jedesmal besucht wird, wenn eine Anweisung verarbeitet werden soll, hat dies zufolge, dass die Ausführung nach Abschluss der aktuellen Befehlsverarbeitung unterbrochen wird.

7. User Program Compile-Script erster Entwurf

Um den Compile-Prozess von Python nach MPY-Byte-Code zu vereinfachen und um diesen in Code::Blocks einzubinden, wurde ein Compile-Script erstellt. Dieses ist in Python geschrieben. Das Compile-Script hat folgende Aufgaben:

1. Aufrufen des MPY-Cross-Compilers und Compilieren des Codes
2. Erfassen der Programmlänge
3. MPY-Byte-Code und Programm Länge in eine Datei schreiben

Das Compile-Script nimmt zu diesen Zeitpunkt als Kommando-Zeilen-Argument den Namen und den Pfad der zu kompilierenden Datei an. Es wird das Python 'tempfile'-Modul verwendet. Dies ermöglicht das MicroPython-User-Programm in ein temporäres Verzeichnis zu schreiben. Dieses wird dort kompiliert. Die daraus entstehende temporäre '.mpy'-Datei wird dann wieder ausgelesen und für die weitere Verarbeitung verwendet.

Der Hintergedanke für den ersten Entwurf ist wie folgt:

1. Durch das Erstellen des temporären Ordners entstehen keine zusätzlichen Dateien, welche später wieder gelöscht werden müssen. Der Nutzer sieht nur die für ihn relevanten Dateien in seinem Projekt-Ordner.
2. Für die Kompilierung sind mehrere Schritte notwendig. Durch die erstellung eines Compile-Scripts wird der Compile-Prozess für den Nutzer vereinfacht.
3. MicroPython muss bei Ausführung auf dem ORB über die Information der Programmlänge verfügen. Die Programmlänge seht dabei immer am Anfang der MPY-Datei. Die Programmlänge wird in den ersten 4 Bytes des Programm-Fash-Bereiches abgelegt. Mit dieser Information kann dann später das Programm korrekt aus dem Programm-Flash ausgelesen werden.

Auch wenn es unter Windows bessere Wege gibt, als die Länge des Programmes immer vor die Datei zu schreiben, so ist dies eine Möglichkeit, dieses Prinzip auszuprobieren. Auch da es sich als eine sinnvolle Lösung für ein später auftretendes Problem anbietet. Denn das ORB speichert das Programm in einem Flash-Bereich ohne fest definierte Programmlänge. Um den MPY-Byte-Code auslesen zu können, muss klar sein, wieviel Bytes aus dem Flash-Speicher ausgelesen werden sollen.

Im späteren Verlauf wurde das User-Compile-Script erweitert. Es wurde im Wesentlichen ein neuer Data-Frame eingeführt. Außerdem wurde das Kompilieren zu Intel-Hex erweitert. Diese Anpassungen finden sich unter 10. User Program Compile-Script vollständig umgesetzt und sind eine direkte Folge aus der Anbindung an den ORB-Monitor, sowie Erweiterungen zur Unterstützung von mehr als einem User-Python-Script. Diese Änderungen machen jedoch erst im Kontext der ORB-Firmware Sinn und werden daher, zum jetzigen Zeitpunkt, nicht genauer erklärt.

8. Testen der Windowsumgebung

Nachdem nun die Implementierung unter Windows, abgesehen von Verbesserungen und Bugfixes, abgeschlossen war, musste als nächstes die Windows-Anwendung getestet werden. Dies war ein einfacher Funktions-Test, um zu Validieren, dass alle ORB-Funktionen korrekt auf Python abgebildet wurden. Das genaue Vorgehen der Tests ist in der Datei Tests,Spezifikationen.md unter Testen in der Code::Blocks-Umgebung nachzulesen.

9. Integration in ORB-Firmware

9.1. Entwicklungsgerüst

Der erste Schritt in der Entwicklung auf dem STM32-F405-Mikrocontroller bestand darin, das grundlegende Entwicklungsgerüst zu erstellen. Dabei ging es zunächst darum, die grundlegende Funktion des MicroPython-Ports zu testen. An diesem Punkt war es noch nicht entscheidend, Programme zu übertragen oder tatsächliche Funktionen zu nutzen.

Zunächst wurde die MicroPython-VM in das ORB-Firmware-Projekt integriert und leere C-Interfaceklassen für die Firmware erstellt. Der auszuführende Byte-Code wurde dabei als statisches Array in die Firmware integriert und gemeinsam mit ihr übertragen. Das erste Testprogramm war ein einfacher Funktionsaufruf, welcher seitens der C-Interfaceklassen eine LED eingeschaltet hat. In diesem Schritt wurde die eigentliche ORB-Firmware zunächst außen vor gelassen. Das bedeutet, dass alle Tasks und zusätzlichen Funktionen der ORB-Firmware auskommentiert wurden. Lediglich die eingebundene EMB-Sys-Lib der ORB-Firmware wurde verwendet, um auf deren LED-Klassen zugreifen zu können.

Nun musste die Firmware auf den Mikrocontroller geflasht werden. Dazu wurde das Tool 'dfu-util' verwendet. Hierzu musste der Mikrocontroller zunächst in den DFU-Modus versetzt werden. Zusätzlich wurde der Mikrocontroller mithilfe des Tools 'zadig' als ein WinUSB-Gerät konfiguriert um die Verwendung von 'dfu-util' zu ermöglichen.

Nachdem diese vorbereitenden Schritte abgeschlossen waren, konnte mit der eigentlichen Entwicklung des Embed-Ports begonnen werden, um diesen schrittweise in die ORB-Firmware zu integrieren.

9.2. Übertragung des Programmes

Um die MicroPython-Funktionen testen zu können, war zunächst die Programmübertragung auf die ORB-Firmware wichtig. Außerdem musste das Ausführen von Programmen direkt aus dem Flash-Speicher realisiert werden. Dies würde die spätere Entwicklungszeit deutlich verkürzen und ermöglichen, effizienter zu testen.

Es wurde entschieden, Sektor 11 als Speicherort für die ORB-Programme zu wählen. Sektor 11 liegt weit entfernt von allen anderen genutzten Speicherblöcken, einschließlich des Firmware-Codes.

Sektor 11 befindet sich an der Adresse 0x080E0000. 'dfu-util' kann Byte-Daten an beliebiger Stelle schreiben. Die MPY-Binär-Datei kann mit dem folgenden Befehl übertragen werden:

```
dfu-util -a address_offset --dfuse-address 0x080E0000 -D <bin_pfad>
```

Die MPY-Binärdatei hatte zu diesem Zeitpunkt folgende Struktur: `<4byte_länge><xxbyte_ mpy-bytes>` . Im späteren Verlauf wurde dieser Daten-Frame erweitert 10.3. Binary-Data-Frame.

Die Länge muss zusätzlich übertragen werden, um zu bestimmen, wie viele Bytes aus dem Flash-Speicher geladen werden müssen. Diese Längenangabe liegt an einer vordefinierten Speicheradresse: 0x080E0000 - 0x080E0003. Die Kodierung dieses 32-Bit-Wertes erfolgt im Big-Endian-Format. Im späteren Verlauf dieses Projektes wird diese Längenadresse weiterhin verwendet, jedoch als Länge des Daten-Frames. Dieser wurde für das Verwenden von mehreren Programm-Modulen entwickelt, vergleichbar mit der 'Middleware' des C++-Projektes.

9.3. Umsetzen der tatsächlichen Modul Funktionen

In diesem Schritt wurden die MicroPython-Module mit den tatsächlichen ORB-Funktionen verbunden. Im Wesentlichen ist dieser Schritt das Umsetzen der C-Interface-Implementationen. Für diesen Zweck werden die durch die AppTask bereit gestellten Funktionen verwendet. Dies sind die gleichen Funktionen welche an die 'orblocal.h' überreicht werden. Für diesen Schritt waren nicht sonderlich viele Änderungen vonnöten. Die Motor-, Sensor-, Servo- und Timer-Funktionen ließen sich durch das Anbinden von einfachen Funktions-Aufrufen realisieren.

Wie etwa für die Funktionen des Time-Moduls: Aus: 'ORB-Python/src/c_interface/implementation/Time_C_Interface.cpp', ab Zeile: 5:

```
extern "C" {
    uint32_t getTime(){
        return AppTask::getTime(nullptr);
    }

    void wait(uint32_t time){
        AppTask::wait(nullptr, time);
    }
}
```

Ein besonderes Vorgehen wurde hier nur bei den Memory- und Monitor-Funktionen benötigt. Für die Memory-Funktionen musste der verwendete Speicher-Block angepasst werden. Dies lag an der wachsenden Firmware-Größe. Ein Nutzer war in der Lage Speicher-Bereiche, die von der Firmware verwendet wurden, zu überschreiben. Aufgrund dessen wurde der von der AppTask verwendete Nutzer-Speicher-Block auf Sektor 10 verschoben. Dies hat zur Folge, dass auch für das C++-Programm der Nutzer-Speicher an dieser Stelle liegt. Das Monitor-Interface musste eine Funktion bereitgestellt bekommen, welche von dem MicroPython-Modul aus aufgerufen werden kann. Da die setMonitorText-Funktion der AppTask eine 'va_list', sowie dessen 'args' erwartet, musste das Monitor-Interface erweitert werden. Diese Erweiterung erlaubt den einfachen Aufruf einer Print-Funktion, ohne das Anpassungen an dem MicroPython-Modul vorgenommen werden müssen.

Aus: 'ORB-Python/src/c_interface/implementation/Monitor_C_Interface.cpp', ab Zeile: 11:

```
void print(BYTE line, const char *format, ...) {
    va_list args;
    va_start(args, format);

    AppTask::setMonitorText(nullptr, line, format, args);

    va_end(args);
}

void setMonitorText(BYTE line, const char *str, size_t len){
    line %= 4;
    print(line, "%.*s", len, str);
}
```

9.4. Umsetzen der Python-Task

Nachdem nun die ORB-Funktionen in die MP-VM eingebunden waren, wurde als nächstes die Python-Task erstellt. Die Python-Task wird von der RTOS::TASK abgeleitet und setzt dessen Funktionen um. Eine RTOS::TASK ist eine Schnittstelle, welche von dem RTOS des ORB verwendet werden kann. Das Registrieren der Task bei dem RTOS geschieht im Constructor der Task und musste hier nicht angepasst werden. Das RTOS des ORB ruft, sofern die Task als laufend gesetzt ist, die Update-Funktion parallel zu den anderen RTOS-Task auf. Außerdem werden 'start'- und 'stop'-Funktionen für diese bereitgestellt. Diese bereitgestellte Logik wurde verwendet um die Python-Task zu realisieren. Im Wesentlichen verwendet die Python-Task die Start-, Stop-, und Update-Logik um die VM-Schnittstelle zu verwalten. Der Workflow der Python-Task wird durch die folgenden Funktionen realisiert:

1. PythonTask::Start(BYTE para)
   Der Erste Aufruf für die Python-Task ist immer der Startbefehl. Hier wird die Python-Task gestartet. Nach diesem Funktions-Aufruf führt das RTOS die PythonTask::Upadte()-Funktion aus.
2. PythonTask::update()
   Die Update-Funktion einer RTOS-Task wird parallel zu den Update-Funktionen der anderen Tasks ausgeführt. Im Fall der Python-Task initialisiert dieser Aufruf die MP-VM und führt den Python-User-Progamm-Code aus.

Am Ende der Python-Task wird die RTOS::Task 'stop'-Funktion aufgerufen. Dies führt dazu, dass der Python-Code immer nur einmal ausgeführt wird.

3. PythonTask::userInterrupt()
   Dieser Funktions-Aufruf wird verwendet, um die Python-Task zu unterbrechen. Die C++-Task verwendet die RTOS::stop-Funktion um die Ausführung zu unterbrechen. Im Gegensatz dazu wird in der Python-Task der Interrupt gescheduled. Dieser Interrupt unterbricht das Python-Programm durch eine Exception, wie in 6.2. MicroPython-Ausführung unterbrechen beschrieben. Die RTOS::update Funktion wird in jedem Fall bis zum Ende ausgeführt und darin inbegriffen die De-Initialisierung der Python-VM.

Entlang dieser 3 Funktionen werden Status-Flags gesetzt, welche verwendet werden, um den aktuellen Zusand der Python-Task auslesen zu können. Hierbei werden im Wesentlichen die VM-Schnittstellen-Status-Funktionen durch die Python-Task eingebunden. Jedoch werden diese auch leicht durch die Python-Task erweitert. Die ORB-Firmware soll immer nur mit den Funktionen der Python-Task arbeiten.

Zusätzlich wird in der PythonTask das Laden des Nutzer-Programmes realisiert. Im Wesentlichen sind dies nur Lese-Operationen auf dem Flash-Speicher des ORB.

Aus: 'ORB-Firmware/src/PythonTask.cpp', ab Zeile: 104:

```
uint8_t* loadProgram(int length) {
    uint8_t* programData = (uint8_t*)malloc(length * sizeof(uint8_t));
    if (programData == nullptr) {
        return nullptr;
    }

    for( DWORD i=0;i<length;i++) {
        programData[i] = programMem.read(PROGRAM_LENGTH_BYTES + LANGUAGE_FLAG_BYTE + i);
    }
    return programData;
}

uint32_t loadProgramLength() {
    return (programMem.read(LANGUAGE_FLAG_BYTE + 0) << 24) |
           (programMem.read(LANGUAGE_FLAG_BYTE + 1) << 16) |
           (programMem.read(LANGUAGE_FLAG_BYTE + 2) << 8)  |
           (programMem.read(LANGUAGE_FLAG_BYTE + 3));
}
```

Dies sind konzeptionell die gleichen Funktionen, welche auch von der Windows-Application umgesetzt werden.

9.5. Anbinden des ORB-Monitor

Als Nächstes war es wichtig den ORB-Monitor mit der Python-Task zu verbinden. Dies ermöglicht das Testen von Text-Ausgaben. Also die Verwendung der 'setMonitor'- sowie 'print'-Funktion. Für diesen Zweck wurde die USB-Task wieder in die ORB-Firmware eingebaut. Im Wesentlichen mussten hier keine größeren Änderungen vorgenommen werden. Jedoch war es in diesem Schritt bereits wichtig die Status-Informationen der VM-Schnittstelle an die ORB-Firmware weiterzugeben. Dies hatte den Grund, das die Funktionen des ORB zurückgesetzt werden, solange keine AppTask läuft. Dies bedeutet, dass die ORB-Monitor-Ausgabe nicht aktuallisiert wird bzw. Motoren etc. werden wieder zurückgesetzt, falls die USB-Task nicht weiß, dass ein User-Programm läuft. Im Wesentlichen musste hier das Verhalten der AppTask für die Python-Task nachgebaut werden. Dies gewährte einen direkten Einblick darauf, an welchen Stellen die ORB-Firmware für eine funktional korrekte Python-Task angepasst werden musste.

9.5.1. Anpassen der Print-Funtkion

Nachdem dies umgesetzt war, ist das nächste Problem aufgefallen. Die ORB-Monitor-Ausgabe geschieht über 4 Zeilen in einem Textfeld. Jede Zeile kann maximal 31 Zeichen enthalten. Aufgrund dessen musste die Print-Funktion, der MP-VM, angepasst werden. Der MicroPython-Embed-Port bietet einen einfachen Weg, Änderungen an der Funktionsweise der Print-Funktion vorzunehmen. In der Datei 'ORB-Python\libs\mp_embed\micropython_embed\port\mphalport.c' gibt es eine Funktion 'mp_hal_stdout_tx_strn_cooked'. Diese wird von dem Embed-Port für die Print-Funktion verwendet. Hier gibt es ein Problem. Die Print-Funktion wird mit Einzelteilen des auszugebenden Strings aufgerufen. Auf dieses Problem wird in der Aussicht unter Ausgabe von Print Anweisungen und Fehler-Meldungen weiter eingegangen. Jedes MicroPython-Print wird mit den Sonderzeichen '\r\n' abgeschlossen. Diese Eigenschaft konnte genutzt werden, um den String auf dem ORB-Monitor auszugeben. Die String-Bestandteile werden zwischengespeichert. Wird in einem String das Zeichen '\r\n' erkannt, führt die 'mp_hal_stdout_tx_strn_cooked' Operationen durch. Diese geben sinngemäß den String aus und bereiten die nächste Zeile der Ausgabe vor. Dies ermöglicht eine verhältnismäßig sinnvolle Ausgabe auf dem ORB-Monitor.

9.5.2. Anpassen der Exception Ausgabe

Ein weiterer Stolperstein ist die Ausgabe von Exceptions. Diese sind oftmals zu groß um sie vollständig auf dem ORB-Monitor auszugeben. Daher wurde sich darauf beschränkt, nicht alle Informationen der Exceptions auszugeben. Es werden nur die nötigsten Informationen über eine Exception an den ORB-Monitor weitergegeben. Namentlich den Exception-Typ und die Exception-Message. So sind zumindest die meisten Exceptions sinnvoll auszugeben. Vor allem vom Nutzer erstellte Exceptions lassen sich so gut verwenden. Da diese den, vom Nutzer, eingestellten Exception-Text weiterhin ausgeben. Der Stack-Trace wird nicht ausgegeben, da dieser die schwer darstellbaren und am ehesten verzichtbaren Informationen enthält. Verwendet man MicroPython-Byte-Code wird ohnehin die Zeile, in der der Fehler aufgetreten ist, nicht angegeben. Jedoch geht durch diese Entscheidung die Ausgabe der Funktion, welche zu dem Fehler geführt hat, verloren.

9.5.3 Handhabung von Hard-Faults

Um einen zusätzlichen Schutzmechanismus in die ORB-Firmware bei Hard-Faults einzubauen, wurde die ORB-Firmware um einen Fault-Handler erweitert. Die RTOS-Tasks stützen sich in ihrer Funktionsweise auf IRQ-Interrupts. Die IRQ-Interrupts müssen unterbrochen werden, um die ORB-Tasks nicht parallel zu dem Hard-Fault-Handler-Code auszuführen. Dieser IRQ-Interrupt-Disable-Befehl ist atomar, also eine nicht kritische Operation:

```
__asm volatile ("cpsid i");
```

[vgl. STM, „3.11.2 CPS“]

Die darauf folgenden Operationen sind das Setzten der GPIO-/Timer-Reset-Register, welche nur ein einfacher Zugriff auf eine vordefinierte Speicher-Adresse in dem Mikrocontroller-Funktion-Register sind. Durch ihre Natur als Reset-Flags wird dieser Zugriff als unproblematisch angesehen. Der aktuelle Zustand des Mikrocontrollers sollte für diese Operationen keine Rolle spielen. Dieses Vorgehen soll bewirken, dass die angeschlossene Peripherie ausgeschaltet wird, bzw. keine neuen Steuer-Befehle bekommt. Der Gedanke hier ist das Schützen von z.B. Servomotoren, welche durch ein fehlerhaftes Programm beschädigt werden könnten. Der Timer-Reset würde an dieser Stelle das von den Servomotoren genutzte PWM-Signal abstellen. Zusätzlich werden alle Status-LEDs des Boards nach dem Reset neu konfiguriert und angeschaltet. Dies informiert den Nutzer, dass es zu einem Hard-Fault gekommen ist. Um diese Funktionalität auf eine sichere Art und Weise testen zu können, kann man in seinem C++-Programm einen Null-Pointer dereferenzieren. Dies produziert einen Mem-Fault.

Dieser Zusatz ist eher für Fehlererkennung und Fehlerhandhabung in der Firmware gedacht, sowie ein Schutz-Mechanismus bei fehlerhafter Verwendung des Boards. Wie z.B. das Aufspielen der falschen Firmware-Version für die verwendete Hardware. Am Ende des Fault-Handlers wird eine Endlos-Schleife ausgeführt, wie es der Default-Fault-Handler ausführen würde. Eine weitere Funktion des Hard-Fault-Handler ist der Umgang mit Fehlern, welche nicht unbedingt durch den Nutzer entstanden sind, wie z.B. Flash-Korruption oder Fehler in der Firmware-Implementierung.

Eine umfangreiche Fault-Handler-Logik bringt Risiken mit sich, da das Programm-Verhalten, sobald ein Fault auftritt, undefiniert ist. Unter Berücksichtigung der oben genannten Punkte wird das Einführen von dieser Logik jedoch als sinnvoll eingestuft.

9.5.4 Zusätzliche Konfigurations-Flags

Wie in 4.2. Compilieren und Ausführen einer MPY-Binär-Datei beschrieben, wird die mit der MP-VM zusammenhängenden Konfigurationen in der 'mpconfigport.h'-Datei erwartet. Die 'mpconfigport.h'-Datei wird erweitert durch: ORB-Python\src\python-vm\orb_config_port.h. Die Platzierung der 'orb_config_port.h' in dem ORB-Python-Projekt ist eine Designentscheidung. Die Konfiguration der MP-VM sollte Teil des ORB-Python-Projektes sein. Diese Datei enthält, so wie vorher die 'mpconfigport.h', alle für die MP-VM benötigten Definitionen. Jedoch wurde die 'orb_config_port.h' auch durch zusätzliche Definitionen erweitert. So werden hier die verschiedenen Language-Flags gesetzt und Exit-Codes definiert. Die Language-Flag ermöglicht das Unterscheiden von C++- und Python-Programmen im Flash-Speicher. Im nächsten Kapitel, 9.6. Wiederherstellen der ORB-Funktionen, wird dies genauer erklärt. Außerdem ist wichtig anzumerken, dass alle Änderungen in dem MicroPython-Projekt, wie z.B. das Einführen der 'Exit()'-Funktion, durch eine Definition an das Projekt angebunden werden.

Aus: 'micropython/py/modbuiltins.c', ab Zeile: 644:

```
#if ORB_EXIT
{ MP_ROM_QSTR(MP_QSTR_exit), MP_ROM_PTR(&mp_exit_obj) },
#endif
```

Die zusätzlichen Konfigurations-Flags haben zwei Funktionen. Erstens wird der Design-Philosophie des MicroPython-Projektes treu geblieben. Möchte man das ORB-Projekt ohne ein bestimmtes Feature kompilieren, etwa für Debug-Zwecke, so kann man es durch das Entfernen der Definition abwählen. In diesem Projekt ist aber ein weiterer Grund ausschlaggebend für diese Entscheidung. Zweitens sollte klar erkennbar sein, welche Änderungen an der MP-VM zu welchem System-Teil gehören. Möchte man wissen, wie z.B. die 'Exit()'-Funktion angebunden ist, so reicht es nach der Definition 'ORB_EXIT' zu suchen. Dies ist ein Versuch, die Änderungen an dem MicroPython-Projekt übersichtlich zu halten.

Das Projekt wurde jedoch nicht nur um selbsterstellte Funktionen erweitert. Im Laufe der Firmware-Implementierung wurde die MP-VM um von MicroPython bereitgestellte Funktionen erweitert. Am wichtigsten sind hier das 'Math'-Modul, 'MIN'- und 'MAX'-Funktionen, sowie das implizite Verwenden des Daten-Typs 'Long', sobald ein Int-Overflow festgestellt wurde. Dies wurde in die Konfiguration aufgenommen, um einen MP-VM Absturz zu vermeiden. Bis zu diesem Zeitpunkt konnten nur 'Short Int' verwendet werden. Diese haben den Nachteil, dass bei einem Overflow die MP-VM mit der Exception "Overflow Error" beendet wird. Dies ist durch die Natur der 'Short Int' als Nicht-Konkrete-MicroPython-Objekte gegeben, da diese wie in 'Konzepte' beschrieben, ihren Datenwert, sowie Typ-Bytes zusätzlich in ihrem Pointer verwalten müssen. Bei einem Overflow könnte diese Besonderheit zu einem Typ-Wechsel führen. Es gibt keine Mechanismen um einen Overflow korrekt abzubilden. Der Daten-Typ 'Long' hingegen ist ein Konkretes-MicroPython-Objekt und verwaltet seinen Wert in einem zusätzlichen Daten-Feld. Somit hat man hier das Overflow-Veralten, welches man aus C/C++ kennt. Um den Unterschied zwischen Nicht-Konkreten-MicroPython-Objekten und Konkreten-MicroPython-Objekten besser zu verstehen, finden sich in 'Konzepte' unter MicroPython-Typ-Klassifizierungen und dem darunter verlinken Kapitel MicroPython Typ Zuordnung Erklärungen.

9.6. Wiederherstellen der ORB-Funktionen

Nun musste die Funktionalität der ORB-Firmware wieder hergestellt werden. Die Python-Task musste vollständig in das User-Interface integriert werden. Das User-Interface hat eine Vielzahl an Funktionen um die AppTask zu verwalten. Diese mussten um das Verwalten der Python-Task erweitert werden. Wie z.B. die isAppActive()-Funktion.

Nach: [ORB-FW, 'ORB-Firmware/src/UserInterface.cpp', ab Zeile: 107]:

```
bool UserInterface::isAppActive()
{
  return( app.isRunning() || pythonTask.isRunning() || pythonTask.isStarting());
}
```

Hier anzumerken ist, dass in diesem Schritt betroffene Funktionen des Unser-Interface namentlich angepasst wurden. Dies soll wiederspiegeln, dass jetzt auch die Python-Task verwaltet wird. Die Anpassung der 'isAppActive'-Funktion führt dazu, dass alle Statusabfragen, die zuvor von der AppTask beantwortet wurden, nun auch von der Python-Task beantwortet werden. Durch diese Entscheidung wurde das Einbinden der Python-Task in die ORB-Firmware zu einem gradlinigen Prozess.

Außerdem wurde die 'Remote'-Klasse angepasst. Diese wird für USB- und Bluetooth-Kommunikation verwendet. Sie wurde erweitert so das sie Exceptions der MP-VM auf dem ORB-Monitor ausgibt. Dies ermöglicht es auch nach Programmbeendigung ohne großen Aufwand die Fehlermeldungen der Python-Task anzuzeigen.

Jedoch gab es hier ein weiteres Problem zu lösen. Bis zu diesem Zeitpunkt wurde nur die Python-Task alleine in der ORB-Firmware ausgeführt. Jedoch muss es eine Möglichkeit geben, sowohl C++- als auch Python-Programme auf das ORB zu übertragen und auszuführen. Es muss also für die ORB-Firmware eine Möglichkeit geben zu erkennen, welche Art Programm zu einem gegebenen Zeitpunkt im Programm-Flash der ORB liegt. Für diesen Zweck wurde eine Language-Flag eingeführt. Die Language-Flag hat eine Länge von einem Byte. Diese befindet sich immer am Anfang des Nutzer-Programm-Speichers. Der Flag-Wert '0b00001111' steht dafür, dass sich im Speicher ein Python-Programm befindet. Der Flag-Wert '0b1111000' steht dafür, dass ein C++-Programm vorhanden ist. Das User-Interface liest dieses Flag bei einem gewünschten Programm-Start. Dadurch kann das User-Interface die Programmausführung entweder an die Python- oder App-Task weiterleiten.

Für diesen Zweck wurde das Binär-Format des Python-User-Programmes angepasst. Zusätzlich musste diese Information in der ORB-Application, also dem C++-Programm vorhanden sein. Für diesen Zweck wurde das ORB-Application-Projekt als Submodule mit in das ORB-Projekt aufgenommen. Hier wurde ein zusätzliches Daten-Segment in der Linker-Datei der ORB-Application eingeführt:

Nach: [ORB-APP, 'ORB-Application/Firmware/Common/Src/Local/stm32f4xx_gcc.ld', ab Zeile: 182]:

```
.flagSegment 0x080E0000:
{
    KEEP(*(.flagSection))
}
.startAppSegment 0x080E0004 :
{
    KEEP(*(.startAppSection))
}
```

Das Beschreiben des 'flagSegment' geschieht in der 'entry.cpp'-Datei:

Aus 'ORB-Application/Firmware/Common/Src/Local/entry.cpp', ab Zeile: 34:

```
__attribute__((section(".flagSection"), used))
const uint8_t flag = 0b11110000;
```

Hier ist zu erkennen, dass wir ein Byte in die 'flagSection' schreiben. Dieser Wert entspricht der C++-Language-Flag und kann durch das User-Interface erkannt werden.

Oben zu erkennen ist, dass auch das 'startAppSegment' verschoben wurde. Dies wurde angepasst, da nun auch der C++-Programm-Flash-Bereich auf Sektor 11 verschoben war. Eine direkte Folge aus den Änderungen durch 9.2. Übertragung des Programmes. Das StartAppSegment wird von der AppTask verwendet um das C++-Programm auszuführen. Die Adresse des Segmentes wird auf eine Funktion gecastet. Diese wird für den Start des Nutzer-Programms ausgeführt:

Nach: [ORB-FW, 'ORB-Firmware/src/AppTask.cpp', ab Zeile: 67]:

```
//*****************************************************************
#if (BOARD_MAIN == 00 && BOARD_SUB == 22)
 unsigned *addr = (unsigned*)0x80E0004;
#elif (BOARD_MAIN == 00 && BOARD_SUB == 30)
 unsigned *addr = (unsigned*)0x80E0004;
#elif (BOARD_MAIN == 01 && BOARD_SUB == 00)
 unsigned *addr = (unsigned*)0x80E0004;
#else
  #error "Board hardware version not defined"
#endif
  int (*func)(BYTE para, cORBlocal &ptr) =  (int (*)(BYTE, cORBlocal &))(*addr);
```

Wie hier zu sehen ist, wurde auch in der AppTask die Funktions-Adresse angepasst. Durch diese Änderungen war es möglich, die Programm-Unterscheidung zu realisieren. Nun konnte man sowohl Python-, als auch C++-Programme von dem ORB ausfühen lassen.

10. User Program Compile-Script vollständig umgesetzt

Wie bereits unter 7. User Program Compile-Script erster Entwurft erwährt wurde das Compile-Script erweitert.

10.1. Unterstützung für Hex-Compilierung

Für die Erweiterung der ORB-Firmware wurde, über den Zeitraum der Entwicklung, der DFU-Prozess zum Aufspielen des User-Programms verwendet. Da die Firmware ohnehin öfter neu aufgespielt werden musste und das ORB somit schon im DFU-Modus war, hat sich dies nicht als Hindernis geäußert.

Für eine nutzerfreundliche Verwendung des ORB ist es jedoch essenziell, den ORB-Monitor als Entwicklungs-Tool im vollen Umfang nutzen zu können. Der ORB-Monitor erwartet eine Intel-Hex, um das C++-Programm auf den ORB übertragen zu können. Dieser Prozess soll auch für Python-Programme genutzt werden. Für diesen Zweck wurde das Compile-Script erweitert. Glücklicherweise gibt es ein Python-Projekt https://github.com/python-intelhex/intelhex welches genau diese Aufgabe erfüllt. Dieses wurde zu Teilen in das Compile-Script eingefügt. Im Wesentlichen hat dies den Compilier-Prozess nicht verändert, sondern nur erweitert. Die im Vorfeld generierte '.bin'-Datei bleibt erhalten. Diese '.bin'-Datei wird weiterhin unter Windows verwendet. Zusätzlich wird diese Binär-Datei zu einem Intel-Hex umgewandelt. Die daraus resultierende Datei hat das Kürzel '.hex'.

10.2. Multi-File-Pre-Compilation

Eine weitere gewünschte Änderung ist das Zulassen von mehr als einem User-Script auf dem ORB. Dies erlaubt das Erstellen und Einbinden von Middleware. Somit würden sich Python-Scripts konzeptionell wie die C++-Applications aufbauen lassen. Zudem hat der Nutzer eine bessere Möglichkeit sein Python-Projekt zu strukturieren.

Wie bereits in 9.2. Übertragung des Programmes und 4.2. Compilieren und Ausführen einer MPY-Binär-Datei beschrieben wurde das Binär-Dateiformat in diesem Schritt weiter angepasst.

Dem Nutzer wird die Möglichkeit gegeben eine '.build'-Datei anzulegen. Es gibt zwei Felder, 'import' und 'main'. Eine solche Datei könnte wie folgt aussehen:

```
{
    "import": ["middleware/dep.py", "middleware/dep2.py"],
    "main": "test.py"
}
```

Die in 'import' aufgelisteten Dateien werden automatisch der Reihe nach importiert und sind so für den Nutzer direkt verwendbar. Hier anzumerken ist, dass kein File-System umgesetzt wurde. Dieser Prozess erlaubt es nicht, in dem Python-Script ein Import-Statement zu verwenden. Jedoch wird der vollständige Inhalt der Dateien in anderen Scripts durch das automatische Laden zur Laufzeit erreichbar. Imports können dabei von anderen Imports abhängig sein, wobei die Reihenfolge des Importierens zu beachten ist. Die 'main'-Datei ist der Einstiegspunkt des Programmes und wird nach dem Import aller anderen Dateien ausgeführt.

Um dies umzusetzen, werden mehrere Binär-Dateien einzeln kompiliert und mit den nötigen Informationen zu einer Binär-Datei zusammen gefasst. Die zusammengefasste Datei kann dann als ein Programm auf das ORB übertragen werden. Sowohl die Windows-Anwendung als auch die ORB-Firmware können das angepasste Format verarbeiten. Die Multi-File-Pre-Kompilation, sowie die Programmausführung kann über das ORB-Util-Tool durchgeführt werden. Für die Ausführung unter Windows sieht der Befehl wie folgt aus:

```
orb-util -b -t win
```

Dies war die letzte Änderung an dem Binär-Format. Wie der vollständige Daten-Frame aussieht, wird im Folgenden beschrieben.

10.3. Binary-Data-Frame

Wie bereits in den vorherigen Kapiteln erwähnt, hat der Binary-Data-Frame eine Entwicklung durchlaufen. Dieser wurde von seiner anfänglichen Form

```
<Bytes(4)        MPY-Size>
<Bytes(MPY-Size) MPY-Data>
```

durch zusätzliche Anforderungen an die ORB-Firmware erweitert und angepasst. Die aus diesen Anpassungen entstandene finale Version des Binary-Data-Frames lässt sich durch folgendes Muster beschreiben:

```
<Data-Frame>
    <Bytes(1) Language-Flag: 0b00001111>
    <Bytes(4) Data-Frame-Size>
    <Bytes(1) Number-Of-Modules>
    <MPY-Size-Region>
        <ForEach-Module-Import>
            <Bytes(4) MPY-Size>
        </ForEach-Module-Import>
        <Bytes(4) MPY-Size-Main>
    </MPY-Size-Region>
    <MPY-Data-Region>
        <ForEach-Module-Import>
            <Bytes(MPY-Size) MPY-Data>
        </ForEach-Module-Import>
        <Bytes(MPY-Size-Main) MPY-Data-Main>
    </MPY-Data-Region>
</Data-Frame>
```

Die Language-Flag wird nach wie vor am Anfang des Data-Frames geschrieben. Diese wird weiterhin, wie in 9.6. Wiederherstellen der ORB-Funktionen beschrieben, verarbeitet. Die Data-Frame-Size und Language-Flag befinden sich immer an der selber Speicher-Adresse. Diese erlaubt der PythonTask das Laden der korrekten Byte-Anzahl aus dem Programm-Flash-Speicher. Der gesamte Data-Frame, mit Ausnahme des Python-Flags, wird an die MP-Schnittstelle weitergegeben. Ein Module-Import ist hierbei optional. Es ist möglich, nur die 'main'-Datei auf das Board zu übertragen. Wie auf dem Muster erkennbar ist, wird direkt nach der Anzahl der Module, die Länge der jeweiligen MPY-Dateien geschrieben. Diese Liste an MPY-Sizes kann verwendet werden um durch den MPY-Data-Bereich zu iterieren. Dies erlaubt das Laden der verschiedenen Python-Files in die Laufzeitumgebung.

11. Firmware-Test

Nachdem nun die Python-Task vollständig in die ORB-Firmware integriert war, musste die erweiterte ORB-Firmware getestet werden. Wie genau getestet wurde, ist in Tests,Spezifikationen.md unter 2. Testen der ORB-Firmware nachzulesen.

12. ORB-Util

Im Laufe des Projektes sind einige Hilfs-Scripts entstanden. Diese haben es vereinfacht, wiederkehrende Aufgaben durchzuführen. So wie das Pre-Kompilieren eines Python-Scripts, das Flashen der Firmware oder das Übertragen eines Programmes. Um ein nutzerfreundlicheres Verwenden zu gewährleisten, werden diese Aufgaben in dem ORB-Util zusammengefasst.

Um das ORB-Util sinnvoll verwenden zu können wird empfohlen, den Ordner 'ORB-Python\tools' zu der Path-Umgebungs-Variable hinzuzufügen. Dadurch kann das ORB-Util mit dem Aufruf 'orb-util' aus einem Python-Programm-Ordner erreicht werden. Es wird eine Vielzahl an Parametern für die Verwendung des ORB-Util bereitgestellt. Diese können mit dem Aufruf `orb-util --help` aufgelistet werden:

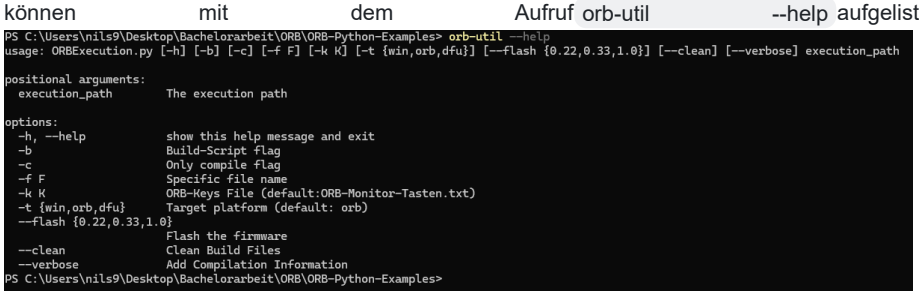

Abbildung 4: ORB-Util Help

Im Wesentlichen sollte diese Help-Page genügen, um das ORB-util zu verwenden. Jedoch gibt es auch ein paar Dinge zu beachten. Das ORB-Util unterstützt nur Python-Programme. Das Kompilieren wird mithilfe des oben beschriebenen '.build' files unterstützt. Der Befehlt `orb-util -b` erwartet, dass sich eine '.build'-Datei in dem Working-Directory befindet. Möchte man diese Build-Datei nicht verwenden, kann mit `orb-util -f <datei_pfad_optional/datei_name>` eine einzelne Datei kompiliert werden. Solange die '-c'-Flag nicht gesetzt ist, wird das Python-Programm für das ausgewählte Target ausgeführt. Im Default-Fall ist dies 'orb', es wird also der ORB-Monitor mit der Kompilerten-Hex aufgerufen. Das 'dfu'-Target und der Flash-Befehl erwarten, dass das ORB im DFU-Modus ist. Es wird das erste Interface, welches das ORB sein könnte, für diesen Prozess verwendet. Das bedeutet, man sollte diese Befehle nur verwenden, wenn das ORB als einziges DFU-Device angeschlossen ist. Für beide Befehle muss der ORB als 'win-usb-devices' konfiguriert sein. Das ORB-Util setzt voraus, dass das ORB-Projekt vollständig eingerichtet wurde. Dies bedeutet, dass die Firmware, die Windows-Application, etc. kompiliert wurden.

Möchte man das ORB-Util verwenden ohne das ORB-Projekt vollständig einzurichten, so wird ein Release angeboten. Dieser enthält alle benötigten Tools und Programme. Der Release enthält eine `README.md`-Datei mit Anweisungen, wie die Release-Version des ORB-Util eingerichtet und verwendet werden kann.

> **Note**
>
> Die Release-Version des ORB-Util kann die Debug-Umgebung stören. Möchte man an diesem Projekt weiter entwickeln, so sollte die Release-Version nicht installiert werden, bzw. vor Verwendung der Entwicklungs-Umgebung, sollte die Release-Verstion aus der Windows-Paths-Variable entfernt werden.

13. Implizites Float, Int und String-Casten

Durch die Natur von C++ ist der implizite Cast von Float-Variablen zu Integer-Variablen erlaubt. Die 'orb_local.h'-Datei verwendet für Zahlenwerte nur Integer in ihren Funktionsaufrufen. Dieses Verhalten sollte auch auf MicroPython abgebildet werden. Die MP-VM hat zu diesem Zeitpunkt keine impliziten Float-Casts unterstützt. Dies wurde durch das Einführen eines Makros geändert. Dieser wird für die MP-VM verwendet, um Zahlenwerte anzunehmen. Der Makro verwendet von MicroPython bereit gestellte Methoden zur Verarbeitung von Zahlenwerten.

Aus: 'ORB-Python/src/python-vm/helper.h', Zeile 54:

```
#define MP_OBJ_GET_NUMBER_VALUE(obj)
        (mp_obj_is_int(obj) ? (float)mp_obj_get_int(obj) : mp_obj_is_float(obj) ?
mp_obj_get_float(obj) : (mp_raise_TypeError(MP_ERROR_TEXT("expected int or float")), 0))
```

Wird ein Zahlenwert erwartet und ein anderer Typ übergeben, wird eine Exception geworfen.

Zusätzlich wurde die Monitor-Klasse erweitert. Diese hat bis zu diesem Zeitpunkt nur die Übergabe von String unterstützt. Ein Aufruf der folgenden Form `monitor.setText(0, sensor.get())` wurde jedoch nicht unterstützt. Der Nutzer musste in diesem Fall den Rückgabewert nach String casten. Die Monitor-Klasse wurde erweitert, um auch nicht String-Objekte als Funktions-Parameter zu erlauben. Für diesen Zweck wird die 'Print'-Funktion des übergebenen MicroPython-Objektes verwendet und auf dem Monitor ausgegeben.

14. Zusätzliche Builtin-Funktionen

Die MP-VM wurde um Buildin-Funktionen erweitert. Dies sind solche Funktionen, welche nicht importiert werden müssen. Die wohl bekannteste dieser Funktionen ist die 'print'-Funktion. Es wurden um 'exit' und 'getArg' erweitert.

14.1 exit

Die Exit-Funktion erfüllt einen ähnlichen Zweck wie der User-Interrupt. Durch den Aufruf der Exit-Funktion wird die MP-VM-Ausführung durch den Python-Nutzer-Code unterbrochen. Dieser Aufruf ist nicht durch Try-Catch-Blöcke abfangbar.

14.2 getArg

Die GetArg-Funktion wird verwendet, um den Startparameter der ORB-Firmware weiterzugeben. Dies wird von der C++-ORB-Application unterstützt. Im Wesentlichen ist die GetArg-Funktion nur die Rückgabe eines Integer-Wertes. Dieser repräsentiert mit welchem Button das ORB-Programm gestartet wurde. Es soll dem Nutzer die Möglichkeit geben z.B. eine Sensor-Kalibrations-Routine mit in sein User-Programm einzubauen.

15. Abschätzen des Speicherbedarfs

Die MP-VM braucht ein Heap-Array. Dieses wird verwendet um die MicroPython-Objekte zu verwalten. Es muss festgelegt werden, wieviel Heap-Speicher dieses Array bekommt. Außerdem muss für die Python-Task eine Stack-Größe eingestellt werden. Es stehen 128 KB RAM zur Verfügung. Der RAM wird sich von Heap und Stack geteilt. Es gibt bereits einen MicroPython-Port für den STM32F405 Mikrocontroller. Aus desser Konfiguration lässt sich folgende Information entnehmen. Es sollten mindestens 2 KB Stack- und 16 KB Heap-Speicher vorgesehen sein. Diese Information wird verwendet, um den eingestellten Heap-/Stack-Speicher abzuschätzen.

```
/* produce a link error if there is not this amount of RAM for these sections */
_minimum_stack_size = 2K;
_minimum_heap_size = 16K;
```

[MP, 'micropython/ports/stm32/boards/stm32f405.ld', ab Zeile: 16]

Ein einfaches MicroPython-Objekt belegt 32 Bit im MicroPython-Heap, also die Größe eines Void-Pointers. Das bedeutet, dass 16 KB Heap-Speicher bis zu 512 MicroPython-Objekte handhaben könnten. Es muss jedoch bedacht werden, dass Methoden-Deklarationen oder komplexere Objekte wie Listen oder Strings, erheblich mehr Speicher verbrauchen als einfache Objekte. Eine Liste mit einem Objekt benötigt beispielsweise 96 Bit, also dreimal so viel wie ein einfaches Objekt. Da der Garbage-Collector gelegentlich während der Programmausführung läuft und wir statische Objektlisten für Motoren, Sensoren etc. verwenden, können etwa 100 MicroPython-Objekte/Werte komfortabel verwaltet werden, ohne das sich Gedanken über die Heap-Nutzung gemacht werden müsste. Im folgenden wird diese Überlegung erläutert.

> Überlegung: Die Motor-/Sensor-Klasse erstellt je nach Funktion komplexere Objekte, wie Listen. Annahme: Wir möchten 50 Motor-Lesewerte speichern.

```
a = motor(0)
b = [a.get()]
for i in range(50):
    b.append(a.get())
print(b)
```

Dies würde 8.736 Bit beanspruchen, also mehr als die Hälfte des verfügbaren Speichers. Dieser Speicher wird auch vom Garbage-Collector ignoriert. Wenn wir diese Logik jedoch in eine Funktion kapseln, sieht dies wie folgt aus:

```
def test():
    a = motor(0)
    b = [a.get()]
    for i in range(50):
        b.append(a.get())
test()
test()
test()
```

Hier funktioniert der Aufruf, da der Garbage-Collector jetzt die lokalen Variablen freigeben kann, auch wenn nicht mehr genügend Speicher übrig sein sollte. Die Verwendung größerer Listen innerhalb einer Funktion ist daher unproblematisch, ebenso wie die Verwaltung einer Liste. Das Handling mehrerer solcher Strukturen ist jedoch nicht praktikabel.

Wenn wir die obigen Informationen betrachten, können wir erkennen, dass etwa 100 MicroPython-Objekte eine realistische Schätzung darstellen. Der Nutzer kann eine Mischung aus komplexen und einfachen Objekten verwenden. 512 Objekte wären zwar theoretisch möglich, doch wenn wir komplexe Objekte wie Listen und Strings berücksichtigen, schrumpft diese Zahl schnell.

Strings, insbesondere Operationen mit Strings, können den Heap-Speicher schnell erschöpfen, da jede String-Operation ein neues String-Objekt erzeugt. Das Anhängen an einen String verdoppelt daher den Speicherbedarf des Strings mindestens bis zum nächsten GC-Zyklus. Da jedoch nicht erwartet wird, dass ein Nutzer mit großen Strings arbeitet, ist dies zu vernachlässigen.

Es wäre sinnvoll, dem Nutzer mindestens 32 KB Heap-Speicher zu gewähren. Dies sollte für die meisten Anwendungen mehr als ausreichend sein, da nach der obigen Einschätzung zwischen 200-1024 MicroPython-Objekte sinnvoll verwaltbar wären. Zusätzlich lässt dies genug Heap-Speicher für die ORB-Funktionalitäten und C++-Application.

Hier wäre es durchaus eine Überlegung wert, der Python-Task aufgrund des größeren Overheads im Vergleich zu einem C++-Programm mehr Heap-Speicher zuzuweisen, etwa 64 KB. Es wurde sich jedoch für 32 KB entschieden, da die Python-Task nicht der C++-Anwendung gegenüber bevorzugt werden soll.

Die maximale Programmlänge eines MicroPython-Programms beträgt derzeit 128 KB. Dies hängt von der Sektorgröße ab, in der das MicroPython-Programm abgelegt wird und sollte für die meisten Anwendungen ausreichend Speicher bieten. Da dieser Sektor von dem C++- und Python-Programm geteilt wird, ist hier keine weitere Einschränkung nötig.

Die Python-Task bekommt 16 KB Stack-Size. Dies liegt deutlich über den Mindestanforderungen und ist in derselben Größenordnung wie die AppTask, die etwa 10 KB erhält. Durch diese Entscheidung verbraucht die Python-Task 48 KB RAM inklusive eines kleinen Overheads zur Verwaltung der MP-VM. Der MP-VM wird also etwa ein Drittel des RAMs zugewiesen. Dies sollte genügend RAM-Speicher für C++-Anwendungen und ORB-Funktionalitäten übrig lassen. Diese Konfiguration konnte durch Tests und die Verwendung des Boards, sowohl für Python- als auch für C++-Programme, validiert werden.

16. Benchmark

Das Benchmarking der ORB-Firmware wurde mit HardWare-Version (im folgenden HW) 0.22 durchgeführt. Die Zeitmessungen des Benchmarking verwenden die ORB-Timer-Funktion. Daher ist in allen Messungen die Verzögerung oder auch Lag der Zeitmessung enthalten. Außerdem beschränkt sich die Zeitmessung auf Millisekunden. Alle errechneten Werte sind daher als gerundete Werte zu betrachten. Es soll hier nur ein grober Vergleich zwischen MicroPython und C++ dargestellt werden. Durch die Verwendung von genügend Berechnungs-Druchläufen, soll die entstandene Ungenauigkeit durch die ORB-Timer-Funktion möglichts wenig ins Gewicht fallen. Die Benchmark-Tests befinden sich unter: ORB-Python-Examples/Benchmark/.

16.1. ORB-Funktionsaufrufe

Das Benchmarking der ORB-Funktionsaufrufe erfolgt über 10.000 aufeinanderfolgende Aufrufe. Es wurden zwei grundlegende Funktionen ausgewählt, die eine direkte Zuordnungen zu C++-Funktionen darstellen: 'getSensorDigital' und 'setMotor'. Zusätzlich wurden zwei Methoden einbezogen, die komplexere Operationen ausführen, wie das Erstellen eines Dictionary(Wörterbuch): 'getSensor' und 'getMotor'. Die Tests wurden bei einer Batteriespannung von ca. 7,9 V durchgeführt.

Function	C++-Pro-Aufruf	Python-Pro-Aufruf	Verhältniss(Python\C++)	Notiz
getSensor	11.3µs	1729.5µs	153.05	Micropyton verwendet an dieser Stelle ein komplexes Element (Dictionary).
getSensorDigital	9.3µs	238.4µs	25.63	
setMotor	12.4µs	292.8µs	23.61	
getMotor	11µs	1089.3µs	99.02	Micropyton verwendet an dieser Stelle ein komplexes Element (Dictionary).
gc.collect	-	5064.2µs	-	Ein C++-Programm muss diese Operation nicht durchführen. Der Nutzer verwaltet selber Speicherzuweisungen.

Abbildung 5: ORB-Funktionsaufrufe

Bei der Betrachtung der Ergebnisse wird deutlich, wie viel Rechenleistung durch die MicroPython-VM verloren geht. Dies ist insbesondere bei der Verwendung komplexer Objekte auffällig.

Beim Benchmarking fällt auf, dass die MicroPython-VM einen Zeitaufwand hinzufügt, der schwer zu quantifizieren ist: die Zeit, die für einen 'gc_sweep'-Zyklus benötigt wird. Diese Operation benötigt deutlich mehr Zeit als ein einfacher Funktionsaufruf. Obwohl der 'gc_sweep' nicht sehr häufig ausgeführt wird, kann dieser Prozess bei Ausführung mehrere Millisekunden in Anspruch nehmen. In den durchgeführten Tests betrug der Zeitaufwand für einen einzelnen 'gc_sweep' etwa 5 Millisekunden. Dieser Wert kann jedoch je nach Heap-Größe, 32 KB während der Tests, den verwendeten Variablen, der Speicherfragmentierung usw. variieren. Angesichts der Tatsache, dass wir in der gleichen Zeit etwa 540 Sensorabfragen in C++ durchführen könnten, ist das eine erhebliche Zeitspanne. Bei der Verwendung von MicroPython für Regelungstechnik könnte dieser 'gc_sweep' einen Regelalgorithmus stören oder zu Oszillationen führen, also zu instabilen Zustand.

16.2. Berechnungen

Es wurden mathematische Verfahren in MicroPython und C++ umgesetzt. Es werden keine Standard-Bibiliotheken verwendet. Beide Programme setzen das gleiche vorgehen um. Die Struktur der Programme ist identisch.

Function	C++-Pro-Aufruf (gerundet)	Python-Pro-Aufruf (gerundet)	Verhältniss(Python\C++)	Notiz
Sinus Annährung	6.04µs	268.33µs	44.41	sin(0-360) in 0.025 Schritten
Quadrat Zahlen	0.7µs	82µs	117.14	(0-9999)^2 in ganzen Schritten
Quadrat-Wurzel(Newton)	499µs	5458µs	10.93	sqrt(0-999) in ganzen Schritten

Abbildung 6: Berechnungen

Hier ist eine interresante Beobachtung zu machen. In jedem Fall ist C++ schneller als MicroPython. Jedoch hängt es hier sehr stark von dem umgesetzen Verfahren ab, wieviel schneller C++ ist. So gibt es Verfahren wie die Quadrat-Wurzel-Berechnung nach Newton, welche im Vergleich zu den bisherigen Tests, nur ~11 mal so langsam ist, wie C++-Implementation. Dies ist, wenn man berücksichtigt, dass der Aufruf einer Sensor-Funktion bereits 25 mal langsamer durch die Verwendung von MicroPython ist, ein überraschendes Ergebnis.

16.3. Filter

Das Filter-Programm verwendet einen NXT-Licht-Sensor. Es wurde eine Variante eines Tiefpass-Filters umgesetzt. Es wurden 1.000 Iterationen für diesen Filter zur Benchmarkmessung eingestellt. Die Iterationen wurden so hoch ansetzen, da bei C++ eine Zeitdifferenz von nur 1 Millisekunde für einen Filteraufruf gemessen wurde. Dies stellt den Minimalwert für Zeitunterschiede dar, der daher nicht zuverlässig ist.

Der Filteraufruf, also die 1.0000 Iterationen, wurde über einen Zeitraum von 5 Minuten wiederholt. Hierfür betrug die durchschnittliche Zeit für einen Filteraufruf für C++ etwa 14 Millisekunden und für MicroPython etwa 2427 Millisekunden. Für diesen Anwendungsfall benötigt das MicroPython-Script somit ungefähr 180 mal so lange wie die C++-Implementierung. Wenn wir einen gefilterten Wert benötigen, beispielsweise für die Regelung eines Systems, ist eine Verzögerung von 1/4 Sekunde wahrscheinlich inakzeptabel. Dagegen könnten 14 Millisekunden, je nach Anwendung, durchaus akzeptabel sein.

Es ist jedoch anzumerken, dass 1.000 Messwerte allein zur Rauschunterdrückung eine hohe Anzahl darstellen. In dem Bereich, in dem so viele Werte benötigt werden, könnte die Entwicklung eines spezielleren Filters eine bessere Lösung darstellen, als einfach die Anzahl der Iterationen zu erhöhen. So könnte je nach Filter-Design MicroPython durchaus effektiv verwendet werden.

16.4. Real World Examples

Bis zu diesem Punkt hat sich das Benchmarken auf den Lag, der durch die Verwendung der MicroPython-VM eingeführt wird, fokusiert. Für komplexe/rechenintensive Aufgaben ist duch das vorhergehende Benchmarking geklärt, dass dies so ist. Jedoch wird hier ein wichtiger Punkt außer Acht gelassen. Die Tasks des ORB, z.b. Sensor Tasks, werden alle `Anzahl_Sensor_Tasks * 200µs` aktuallisiert, also alle 800µs. Außerdem haben Sensoren und das Verwenden von Motoren selber auch eine Verzögerung. Die folgenden Tests sollen für einfache Anwendungen klären, ob die eingeführten Verzögerungen bemerkbar bzw. messbar sind.

16.4.1. Line Follower

> **Note**
>
> Der Line Follower, sowie der Line Follower Smooth, verwendet einen Makeblock Line Follower V2.3 Sensor. So wie 2 Makeblock Motoren.

Es wurde ein Line Follower umgesetzt. Es sollte die Laufzeit des Roboters durch eine Strecke verglichen werden. Bzw. die maximal einstellbare Geschwindigkeit.

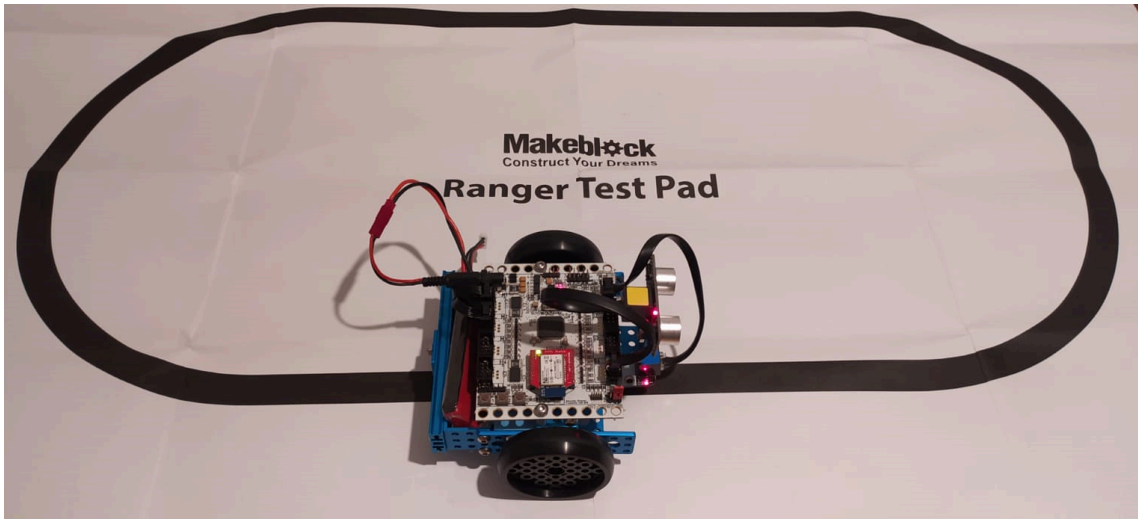

Abbildung 7: Strecke

Zuerst war dies eine sehr einfache Anwendung, welche ein Motor ein- bzw. ausschaltet, in Abhängigkeit zu den Sensor-Mess-Werten. Es sollte getestet werden, wie schnell man den Roboter durch Kurven fahren lassen kann, ohne das er eine Linie verliert. Dies hat sich jedoch als unpraktisches Programm für das Benchmarking herausgestellt. Der Roboter ist sehr unsauber durch Kurven gefahren. Es gab schlagartige Richtungsänderungen. Sowohl C++ als auch Python konnten hier gleich schnell durch die Strecke fahren. Da dies eher davon abhängig war, wie sauber der Roboter sich zu der Linie orientierte. Man musste eher Glück haben, dass der Roboter nach einer Richtungsänderung korrekt platziert war. Hier konnte keine sinnvolle Aussage getroffen werden. Daher wurde der Line Follower um Smothing erweitert.

16.4.2. Line Follower Smoth

Dies ist die aus Line Follower entstandene Variante. Der Roboter schaltet seine Motoren nicht mehr vollständig ab, wenn die Linie verloren wurde, sondern hat einen Übergang von 100%-0%. Durch diesen Geschwindigkeitsübergang fährt der Roboter sauber durch Kurven. Der Roboter wird hier nicht mehr nach Geschwindigkeit verglichen, sondern nach der maximal möglichen Verzögerung für den 100%-0%-Verlauf. Die eingestellte Geschwindigkeit ist für C++ und MicroPython gleich. Die gewählte Geschwindigkeit ist durch den 'SPEED_MODE' mit 800 als Wert gegeben. Hier ist noch wichtig anzumerken, dass die Geschwindigkeiten der beiden Motoren von einander abhängen. Wird die Geschwindigkeitsvariable eines Motors auf 0% geregelt, ist die tatsächlich verwendete Geschwindigkeit eines Motors jedoch immer noch als ein Bruchteil der Geschwindigkeit des anderen Motors eingestellt. Das bedeutet solange ein Motor läuft, wird der andere nie vollständig abgeschaltet. Der Roboter bleibt nur stehen, wenn beide Motoren auf 0% geregelt wurden. Das C++-Programm konnte eine Verzögerung von 95ms für den vollständigen Übergang gewährleisten. Während das MicroPython-Programm den Roboter nur bis zu 90ms Verzögerung erfolgreich durch die Strecke fahren lässt. Die Gesammtdifferenz von 5ms für diesen Reaktions-Zeit-Test ist jedoch überraschend. Das C++-Programm reagiert nur in etwa 5.6% schneller, als das MicroPython-Gegenstück. Dies ist ein Unterschied der zu vernachlässigen ist, besonders unter Berücksichtigung, dass beide Programme nahezu 100ms Reaktions-Zeit haben. Für einen Nutzer wird dies fast nie einen Unterschied machen. Dies zeigt, dass Rechenintensive Programme zwar stark durch die Verwendung von MicroPython eingeschränkt werden, es jedoch auch Programme gibt, die kaum beeinträchtigt werden oder lediglich einen nahezu unbemerkbaren Nachteil aufweisen.

16.4.3. Forwared-Backward

> **Note**
>
> Es werden Lego-NXT-Motoren, sowie zwei NXT-Licht-Sensoren verwendet. Der Roboter wird auf einer schwarzen Linie platziert. Erkennt der Roboter eine schwarze Linie, so fährt er für 250ms zurück. Danach fährt er wieder vorwärts. Dieses Verhalten wird über eine Zeitspanne von 20 Minuten wiederholt. Es wird gezählt, wie oft der Roboter eine Linie erkennt. Der Versuch wurde zweimal wiederholt.

Der Lag wird als Unterschied der idealen Laufzeit zu der tatsächlichen Laufzeit berechnet: (Idealen_Laufzeit - (250ms * 2 * Gezählte_Linien)) / Gezählte_Linien = Lag_Pro_Linie Wobei 250 ms * 2 also die Verzögerung von 250ms vorwärts und rückwärts, der Richtwert für ein idealen Durchlauf ist.

Durchlauf	C++-Linien-Erkannt	Python-Linien-Erkannt	C++-Lag (gerundet)	Python-Lag (gerundet)	Python\C++-Linien-Erkannt-Verhältnis	Python\C++-Lag
1	2398	2318	0.42ms	17.68ms	0.967	42,10
2	2397	2316	0.63ms	18,13ms	0.966	28,78

Abbildung 8: Forwared-Backward

Selbst eine einfache Anwendung wie das Vor- und Zurückfahren bis zu einer Linie, hat einen erkennbaren Unterschied zwischen C++ und MicroPython. Es ist jedoch zu beachten, dass hier keine halben Durchläufe gezählt werden können. Daher sollten alle gemessenen und berechneten Angaben mit Vorsicht interpretiert werden. Betrachetet man das Gesammtergebniss, so ist der Unterschied verhältnissmäßig klein. Betrachtet man wie oft die Linie erreicht wurde, ist MicroPython etwa 3-4% langsamer als C++. Dies ist kein sonderlich großer Unterschied. Betrachtet man jedoch den Unterschied in Verzögerung, ist ein sehr großer Unterschied zwischen Python und C++ erkennbar. Der Python-Code ist ca. 35 mal langsamer als der C++-Code. Dies stimmt mit denen in 1.6.1. ORB-Funktionsaufrufe gemessenen Unterschieden ungefähr überein.

16.5. Schlussfolgerung

Es gibt eine Vielzahl an Performance-Unterschieden, die sich aus der Verwendung der MP-VM ergeben. Je nach Anwendung können Berechnungen vergleichsweise schnell ablaufen, etwas langsamer sein oder signifikant an Geschwindigkeit verlieren. Dieses Verhalten ist gut bei 1.6.2. Berechnungen zu beobachten.

Einige Anwendungsbereiche sind möglicherweise ungeeignet für die Ausführung in der MP-VM. Insbesondere wenn eine hohe Rechenleistung erforderlich ist, z. B. bei 1.6.3. Filtern. Andere Anwendungen werden durch die MP-VM nur leicht beeinträchtigt, was beispielsweise bei 1.6.4.3. Forward Backward beobachtet werden kann.

Es gibt jedoch auch Fälle, in denen die Performance unter MicroPython nahezu identisch zur C++-Implementierung bleibt. Ein Beispiel ist das Benchmarking eines Line Followers: Obwohl ORB-Funktionsaufrufe unter MicroPython mindestens 25-mal langsamer sind als in C++, weist der in MicroPython geschriebene Line Follower Smoth eine mit C++ vergleichbare Leistung auf. Hier zeigt sich eine interessante Eigenschaft: Die Verzögerung des Systems gleicht in manchen Fällen die durch die MicroPython-VM verursachte Verzögerung aus, sodass das Gesamtsystem vergleichbar mit dem C++-Gegenstück ist.

Zusammengefasst hängt der Performance-Unterschied zwischen C++ und Python stark von der jeweiligen Anwendung ab. Dennoch weist das C++-Programm in jeder Anwendung eine bessere Performance auf, auch wenn dieser Vorteil in manchen Fällen kaum spürbar ist. Dies ist jedoch angesichts der Tatsache, dass MicroPython eine virtuelle Maschine als Interpreter verwendet, zu erwarten.

Konzepte

Inhaltsverzeichnis

Warum den MicroPython Embed Port verwenden?

Im Rahmen dieser Bachelorarbeit wurden zwei mögliche Ansätze für die Integration des MicroPython-Interpreters in die ORB-Firmware identifiziert. Der erste Ansatz basiert auf der Verwendung eines der MicroPython-STM-Ports. Dafür könnte man z.B. den ADAFRUIT_F405_EXPRESS-Port wählen. Dieser Ansatz würde die MicroPython-Umgebung als Grundlage nutzen. Somit müsste die ORB-Firmware als Erweiterung des MicroPython-Port eingebunden werden. Allerdings kann man diesen Weg als eher unpraktisch ansehen. Da er Komplikationen im Build-Prozess mit sich bringen könnte. Möglicherweise wären auch umfangreiche Anpassungen an dem MicroPython-Port vorzunehmen, gerade durch die Natur der ORB-Firmware als C++-Projekt.

Die ORB-Firmware sollte idealerweise nur an den notwendigsten Stellen angepasst werden. Wie bereits im Exposé beschrieben, ist das Ziel dieser Bachelorarbeit, den MicroPython-Interpreter als eigenständige Komponente in die ORB-Firmware einzubinden. Somit ist der Ansatz, die ORB-Firmware in das MicroPython-Projekt einzubinden, nicht das erwünschte Vorgehen.

Der zweite Ansatz ist die Verwendung des MicroPython-Embed-Ports. Dieser Ansatz bietet einen klaren Vorteil. Der MicroPython-Interpreter kann als eigenständige Komponente in die ORB-Firmware eingebunden werden und dies, ohne dass die ORB-Firmware stark verändert werden muss. Es müssen lediglich sinnvolle Schnittstellen definiert werden. Dies wären C++-Schnittstellen um die MicroPython-VM bereit zu stellen. So wie Schnittstellen für die ORB-Firmware, welche die C++-Funktionen der ORB-Firmware auf C-Funktionen abbilden sollen und dadurch verwendbar im MicroPython-C-Code machen.

Gleichzeitig hat dieser Ansatz den Vorteil, dass der MicroPython-Embed-Port auch unter Windows kompiliert werden kann. Hier kann mit Hilfe von Code::Blocks oder einer anderen Entwicklungs-Umgebung eine Debug-Umgebung geschafft werden. Mit dieser Debug- und Entwicklungs-Umgebung sollte der Prozess zur Entwicklung für MicroPython bedeutend erleichtert werden.

Aufgrund der genannten Vorteile wird der zweite Ansatz für das Umsetzen dieses Projektes verwendet.

MicroPython-Types

Es gibt eine Vielzahl an MicroPython-Typen die abgebildet werden können. Diese werden in die Obergruppen Konkrete-Typen und Nicht-Konkrete-Typen (none-concrete-types) eingeteilt. Die Konkreten-Typen (concrete-types) sind gerade für die Entwicklung eines MicroPython-Ports interessant. Diese erlauben das Abbilden von Modulen, Klassen und Objekten. Im Folgenden werden diese genauer erklärt. Unter MicroPython-Typ-Klassifizierungen und MicroPython Typ-Zuordnung wird auf ihre Unterschiede in Aufbau und Verwendung genauer eingegangen.

- MicroPython-Object-Type
- MicroPython-Typ-Klassifizierungen
  - Nicht-Konkrete-Typen
  - Konkrete-Typen
    - Module-Type
    - Klassen-Type
      - Selbst definierte Klassen
      - Bereitgestellte Funktionen und Konzepte
  - Funktions-Objekt-Definition

MicroPython-Object-Type

Ein MicroPython-Objekt (mp_obj_t), ist ein abstrakter Zeiger. Dieser wird genutzt, um verschiedene Objekttypen generisch zu behandeln.

> **Note**
>
> MicroPython verwaltet nahezu alles als MicroPython-Objekt. Wird von einem MicroPython-Objekt gesprochen, so handelt es sich um einen durch 'mp_obj_t' abstrahierten Daten-Typ. Es werden selbst an ein Objekt gebundene Funktionen oder Module als MicroPython-Objekte verwaltet.

`mp_obj_t` ist wie folgt definiert:

```
// This is the definition of the opaque MicroPython object type.
// All concrete objects have an encoding within this type and the
// particular encoding is specified by MICROPY_OBJ_REPR.
#if MICROPY_OBJ_REPR == MICROPY_OBJ_REPR_D
typedef uint64_t mp_obj_t;
typedef uint64_t mp_const_obj_t;
#else
typedef void *mp_obj_t;
typedef const void *mp_const_obj_t;
#endif
```

[MP, 'micropython/py/obj.h', ab Zeile: 37]

Da die 'MICROPY_OBJ_REPR'-Definition in diesem Projekt durch die Standardkonfiguration vorgegeben ist, wird 'MICROPY_OBJ_REPR_A' verwendet.

```
#ifndef MICROPY_OBJ_REPR
#define MICROPY_OBJ_REPR (MICROPY_OBJ_REPR_A)
#endif
```

[MP, 'micropython/py/mpconfig.h', ab Zeile: 153]

Damit wird `mp_obj_t` als Alias für einen `Void-Pointer`, also einen typfreien Zeiger verwendet. Aus dem Kommentar zu 'MICROPY_OBJ_REPR_A' lässt sich eine weitere Erkenntnis ableiten.

```
// A MicroPython object is a machine word having the following form:
//  - xxxx...xxx1 : a small int, bits 1 and above are the value
//  - xxxx...x010 : a qstr, bits 3 and above are the value
//  - xxxx...x110 : an immediate object, bits 3 and above are the value
//  - xxxx...xx00 : a pointer to an mp_obj_base_t (unless a fake object)
#define MICROPY_OBJ_REPR_A (0)
```

[MP, 'micropython/py/mpconfig.h', ab Zeile: 112]

Der MicroPython-Objekt-Pointer wird nicht ausschließlich als Void-Pointer genutzt. Er enthält auch Informationen über das gespeicherte Objekt. In einigen Fällen, wie bei 'Small Int', wird der Variablen-Wert im Pointer gespeichert. Zusätzlich werden die hier beschriebenen Bits später für Typ-Zuordnung verwendet.

> Note
>
> Mehr dazu unter MicroPython-Typ-Zuordnung.

Da jetzt klar ist, was der MicroPython-Objekt-Typ ist, kann nun geklärt werden, welche Rolle er spielt und wo und wie er verwendet wird.

Funktionen, die in MicroPython geschrieben werden und auf MicroPython-Objekten arbeiten, erwarten Argumente stets als 'mp_obj_t'. Dazu gehört auch das Self-MicroPython-Objekt, wie in folgendem Beispiel:

```
static mp_obj_t get_button(mp_obj_t self_id) {
    button_obj_t *self = MP_OBJ_TO_PTR(self_id);
    bool ret = getSensorDigital(self->id);
    return mp_obj_new_bool(ret);
}
```

Auch der Rückgabewert einer Methode ist immer vom Typ `mp_obj_t`. Selbst spezielle Rückgabewerte wie None, True oder False sind als solche Objekte gekapselt.

```
#define mp_const_none (MP_OBJ_FROM_PTR(&mp_const_none_obj))
#define mp_const_false (MP_OBJ_FROM_PTR(&mp_const_false_obj))
#define mp_const_true (MP_OBJ_FROM_PTR(&mp_const_true_obj))
```

[MP, 'micropython/py/obj.h', ab Zeile: 890]

> Note
>
> Das Makro `MP_OBJ_FROM_PTR` wird verwendet, um einen beliebigen Pointer zu einem `mp_obj_t` zu casten.

Es gibt bei der Verwaltung von MicroPython-Objekten ein paar Unterschiede. So werden "Vollwertige" Integer als Objekte verwaltet und haben ein 'mp_obj_int_t-struct'. Wie oben beschrieben, stellen jedoch 'Small Int' eine Sonderrolle dar. Der Unterschied im Zugriff lässt sich an der folgenden Methode gut erkennen:

```
mp_int_t mp_obj_int_get_truncated(mp_const_obj_t self_in) {
    if (mp_obj_is_small_int(self_in)) {
```

```
        return MP_OBJ_SMALL_INT_VALUE(self_in);
    } else {
        const mp_obj_int_t *self = self_in;
        return self->val;
    }
}
```

[MP, 'micropython/py/objint_longlong.c', ab Zeile: 284]

Daten-Typen wie 'Small Int' verhalten sich dabei konzeptionell wie jedes andere MicroPython-Objekt. Sie werden genauso wie jedes anderes MicroPython-Objekt als Pointer verwaltet, haben jedoch einen Unterschied bei dem Zugriff auf Daten-Werte.

Im Gegensatz dazu: Arbeitet man mit selbst implementierten oder bereitgestellten Objekten, so sieht der Zugriff in der Regel wie folgt aus:

```
<...>
static void mp_funktion(..., mp_obj_t obj_input, ...){
button_obj_t *button = MP_OBJ_TO_PTR(obj_input);
<...>
```

Man hat ein 'Struct' welches das Objekt beschreibt. In diesem Fall 'button_obj_t'. Mithilfe des Makros 'MP_OBJ_TO_PTR' wird aus dem 'mp_obj_t' ein Pointer zu dem eigentlichen 'Struct'. Dieses kann dann in Folge-Operationen verwendet werden.

> **Note**
>
> Wobei das Makro 'MP_OBJ_TO_PTR' ein einfacher Cast ist. Es ist keine komplexe Operation und eher syntaktischer Zucker:
>
> ```
> #define MP_OBJ_TO_PTR(o) ((void *)(o))
> ```
>
> [MP, 'micropython/py/obj.h', Zeile:316]
>
> Angesichts der Komplexität des MicroPython-Projektes ist eine solche Vereinfachung eine große Hilfe und erleichtert vor allem die Lesbarkeit von MicroPython-C-API-Code. 'MP_OBJ_TO_PTR' ist das Gegenstück zu dem am Anfang erwähnten 'MP_OBJ_FROM_PTR'.

Es ist wichtig, diesen Cast nur vorzunehmen, wenn Typ-Sicherheit gegeben ist. MicroPython bietet jedoch auch eine Möglichkeit für Typ-Prüfungen an. Der Typ-Vergleich eines MicroPython-Objektes setzt voraus, dass dieses korrekt aufgebaut wurde. Es ist wichtig zu beachten, dass die 'mp_obj_base_t' eines MicroPython-Objektes an erster Stelle der Objekt-Struktur seht. Mehr dazu unter Wie funktioniert die Typ-Zuordnung (Konkrete-Typen).

MicroPython-Typ-Klassifizierungen

Im Folgenden werden die verschiedenen Arten der MicroPython-Typen erläutert. Dies sind alles Ausprägungen des abstrakten MicroPython-Objektes. Nicht-Konkrete-Typen (none-concrete-types) sowie Konkrete-Typen (concrete-types) stellen dabei Ober-Klassifizierungen dar. Diese werden im Folgenden kurz qualitativ erklärt.

Nicht-Konkrete-Typen

Nicht-Konkrete-Typen sind solche Typen, welche kein eigenes Typ-Struct haben. Sie verwalten ihre Informationen als Teil ihres `mp_obj_t` pointers. Dabei sind sie kein Zeiger auf ein Strukt, sondern im wesentlichen, ähnlich wie ein einfach Integer, ein Daten-Wert-Träger. Bei diesen wird ein Teil des `mp_obj_t` -pointers in Bereiche für Daten-Werte und Typ-Informationen aufgeteilt. Darunter zählen zum Beispiel 'Small Ints'.

```
// as its first member (small ints, qstr objs and inline floats are not concrete).
```

[MP, 'micropython/py/obj.h' Zeile:52]

Wie Nicht-Konrekte-Typen aufgebaut sind, ist in Zusammenhang mit der Typ-Zuordnung am besten zu verstehen. Dies kann unter MicroPython Typ Zuordnung nachgelesen werden.

Konkrete-Typen

Konkrete-Typen sind MicroPython-Objekte, welche ein eigenes Typ-Struct besitzen. Sie müssen zu diesem gecastet werden und erlauben somit das Verwalten von komplexen Objekten. Zu den Konkreten-Typen gehören Module, sowie Klassen und Objekt-Instanzen. Konkrete-Typen erwarten, wie bereits erwähnt, eine 'mp_obj_base_t' an erster Stelle ihres 'mp_obj_t'-pointers.

```
// Anything that wants to be a concrete MicroPython object must have mp_obj_base_t
```

[MP, 'micropython/py/obj.h' Zeile:53]

Diese Base und der dazugehörige Typ werden verwendet, um das MicroPython-Objekt zu verwalten. Die MP-VM ist darauf angewiesen, dass dieser Typ korrekt konfiguriert ist. So wird dieser z.B. dafür verwendet, eine Objekt-Instanz ihren zugehörigen C-Funtionen zuzuordnen. Ein typloses Objekt, bzw. eines mit falscher Typ-Information, führt zu einem undefinierten Verhalten. Im besten Fall kommt es zu einem Speicherzugriffsfehler, im schlechtesten Fall läuft das Programm fehlerhaft weiter.

Im Folgenden werden ein paar von MicroPython bereitgestellte Konkrete-Typen genauer betrachtet.

Module-Type

Module werden mithilfe des Daten-Typ 'mp_obj_module_t' abgebildet. Im einfachsten Fall setzen sie sich aus zwei Komponenten bzw. Definitionsschritten zusammen.

1. Erstellung eines Dictionaries, das zur Verwaltung von Modul-Meta-Informationen verwendet wird. Beispielsweise werden Modulname und die Init-Funktion, die beim Import des Moduls aufgerufen wird, definiert.

   ```
   static const mp_rom_map_elem_t devices_globals_table[] = {
       { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_devices) },
       { MP_ROM_QSTR(MP_QSTR___init__), MP_ROM_PTR(&devices___init___obj) },
       { MP_ROM_QSTR(MP_QSTR_motor),  MP_ROM_PTR(&motor_type) },
   };
   static MP_DEFINE_CONST_DICT(devices_globals, devices_globals_table);
   ```

   Solche Dictionaries bestehen in der Regel aus einem 'MP_ROM_QSTR'- und 'MP_ROM_PTR'-Paar. Dies ist eine String-Zuordnung zu einem Befehl oder einer Funktionalität und das an diesen String gebundene MicroPython-Objekt. Dies kann auch eine als MicroPython-Objekt gekapselte Funktion, Konstante oder ein Modul sein. Diese werden als 'MP_ROM_PTR' in einen Dictionary-Eintrag eingebunden. Im Wesentlichen ist dies ein Makro, welches MicroPython-Objekte vor einem Garbage-Collector-Clean schützt. Im Falle von 'MP_ROM_QSTR(MP_QSTR___name__)' handelt es sich um einen Sonderfall. Dies ist eine in MicroPython eingebaute Funktionalität. Dem Modul wird der Name 'devices' zugewiesen.

> Dies ist nicht der Import-Name, sondern der Modul-Interne-Name, den das Modul für sich selbst verwaltet. Wie zum Beispiel für die Print-Funktion. So würde `print(time)` den hier für das Time-Modul definierten Namen ausgeben.

Es ist wichtig, solche Namen stets als sogennante QString mit Hilfe von MicroPython-Makros zu generieren.

> Unter QString findet sich eine qualitative Erklärung zu diesen.

Zuletzt muss das Makro `MP_DEFINE_CONST_DICT` verwendet werden. Dieses wandelt `mp_rom_map_elem_t` zu einem MicroPython-Dictionary-Objekt um.

2. Als nächstes muss das MicroPython-Dictionary-Objekt unter einem Modul registriert werden. Hierfür gibt es den vorher erwähnten Typ `mp_obj_module_t` . Dieser hat an dieser Stelle zwei wichtige Attribute.

```
const mp_obj_module_t devices_module = {
 .base = { &mp_type_module },
 .globals = (mp_obj_dict_t *)&devices_globals,
 };
 MP_REGISTER_MODULE(MP_QSTR_devices, devices_module);
```

Die `base` ist ein Pointer, welcher die Typ-Zuordnung für ein MicroPython-Objekt ist. Mehr dazu unter MicroPython Typ Zuordnung. Die Modul-Base ist hier `&mp_type_module` . Jedes Modul hat diesen Pointer als Base. Durch diese Zuweisung verwaltet die MP-VM dieses Objekt als ein Modul. Es erhält somit alle Funktionalitäten und Eigenschaften die man von einem Modul erwarten würde. Das Globals-Attribut umfasst die durch das Modul zugänglichen Klassen, Funktionen, usw. kurz gesagt das oben definierte Dictionary. `MP_REGISTER_MODULE` registriert das Modul, hier wird ein QString angegeben, welcher den Namen des Moduls vorgibt. Dieser Name wird auch für den Import verwendet.

Der Datei micropython/py/makemodulesdef.py ist zu entnehmen, das es drei Arten gibt, Module zu registrieren.

> `makemodulesdef.py` ist ein Python-Script welches im MicroPython-Build-Prozess verwendet wird

MP_REGISTER_MODULE Ein Modul als*Builtin*-Modul deklarieren.**MP_REGISTER_EXTENSIBLE_MODULE**Dieses Modul soll vom Dateisystem aus erweitert werden können.

> Da kein Dateisystem implementiert wird, ist dieser Punkt uninteressant.

MP_REGISTER_MODULE_DELEGATION Wird verwendet, um die Registrierung oder Initialisierung eines Moduls an eine externe Funktion zu delegieren.

> Da die umzusetzenden Funktionen klar definiert sind, spielt diese Modulregistrierung für dieses Projekt keine Rolle.

Klassen-Type

1. Selbst definierte Klassen
2. Bereitgestellte Funktionen und Konzepte

Selbst definierte Klassen

Die Besonderheit von Klasen-Typen im Gegensatz zu Modul-Typen ist, dass sie ihren Typ-Pointer selber verwalten. Ein Klassen-Typ hat immer eine Definition für den Typ-Pointer in dieser Form:

```
const mp_obj_type_t <typ_name>;
```

Die Adresse dieses Pointers ist nach Initialisierung eindeutig und einmalig, solange diese nicht fälschlicherweise verändert wird. Mehr dazu unter MicroPython Typ Zuordnung. Dieser Pointer wird verwendet, um Objekte diesem Typ zuzuordnen. Dies entspricht dem `mp_type_module` Modul-Pointer. Hier wird die erste Parallele in der Verwaltung von Objekten und Modulen klar.

Zusätzlich zu diesem Typ-Pointer kann eine selbst definierte Struktur erstellt werden. In der Form:

```
typedef struct _<struct> {
    mp_obj_base_t base;
    <
    Frei wählbare Felder.
    >
} <struct>_t;
```

Dieses 'Struct' wird später an Objekt-Instanzen gebunden. In diesem 'Struct' können Informationen geschrieben werden, die man für die Verwaltung eines Objektes braucht. Das MicroPython-Objekt wäre hier eine abstrakte Referenz auf diesen Daten-Typ.

Wichtig ist das diese 'Structs' immer über eine Objekt-Base verfügen. Alle anderen Felder sind jedoch frei definierbar.

In diesem Projekt werden an ein Port gebundene Objekte gesondert behandelt. So wie z.B. Motoren, welche nur an Port 1 bis 4 angeschlossen werden können. Es wird zusätzlich ein statisches Objekt-Array erstellt, welches die zu verwaltenden MicroPython-Objekte einmalig anlegt. So sind alle MicroPython-Objekte welche an einen Port gebunden sind, geteilte Objekte. Dies soll den MicroPython-Heap-Verbrauch minimieren und zu einem konsistenten Verwalten von angeschlossenen Geräten führen. Mehr dazu unter Reduzierung des MicroPython-Heap-Verbrauchs durch Objektreferenzen.

Wie oben beschrieben verwaltet das 'Struct' eines Objektes Informationen über dieses. Wie zum Beispiel den Port, mit dem eine Sensor-Instanz initialisiert wurde.

MicroPython-Funktionen, welche an Objekt-Instanzen gebunden werden, habe eine Besonderheit. Sie erhalten das MicroPython-Objekt, welches sie selber sind, als erstes Argument einer jeden Funktion. Dies ist vergleichbar mit dem `self`-Konzept aus der Python-Programmierung.

Hat man also eine Funktion einer Klasse:

```
static mp_obj_t get_button(mp_obj_t self_id) {
    button_obj_t *self = MP_OBJ_TO_PTR(self_id);
    bool ret = getSensorDigital(self->id);
    return mp_obj_new_bool(ret);
}
```

So ist das erste Argument, welches der Funktion mitgegeben wird das `self`. Also ein `mp_obj_t`, welches auf das korrekte 'Struct' gecastet werden kann. Dieses wird dann verwendet, um Operationen in Relation zu dem Objekt-Zustand durchzuführen. Es sollten also alle instanzspezifischen Informationen eines MicroPython-Objektes in dessen Strukt verwaltet werden.

Bereitgestellte Funktionen und Konzepte

MicroPython bietet von sich aus auch verschiedene Objekt-Typen an, wie z.B. Listen oder Dictionaries. Diese sind konzeptionell dasselbe wie eine Objekt-Instanz einer selbst definierten Klasse. Dafür stellt MicroPython vordefinierte Modul-Klassen und Typ-Strukturen bereit. Diese setzen die MicroPython-Objekte auf gleiche Art und Weise um, wie selbstdefinierte Klassen.

```
typedef struct _mp_obj_list_t {
    mp_obj_base_t base;
    size_t alloc;
    size_t len;
    mp_obj_t *items;
} mp_obj_list_t;

void mp_obj_list_init(mp_obj_list_t *o, size_t n);
mp_obj_t mp_obj_list_make_new(const mp_obj_type_t *type_in, size_t n_args, size_t n_kw,
const mp_obj_t *args);
```

[MP, 'micropython/py/objlist.h', ab Zeile: 31]

Dadurch ist es möglich, MicroPython-Objekte oder Module auf gleiche Art und Weise zu erweitern, wie man es bei einem selbstdefinierten Datentyp machen würde. Zum Beispiel könnte man das hier gezeigte MicroPython-Listen-Objekt um eine Get-Index-Funktion erweitern.

Es gibt auch eine Vielzahl an anderen Konzepten die MicroPython-Objekte umsetzen. Aus dem oben angegeben Code-Schnipsel lassen sich zwei dieser Konzepte herauslesen. Der `new`-Operator, hier durch die Funktion `mp_obj_list_make_new` abgebildet. Er enthält immer den Typen des gewünschten Elements als erstes Parameter, hier angegeben als `mp_obj_type_t *type_in`. Das ist der in Klassen-Type beschriebene MicroPython-Objekt-Typ-Pointer. Außerdem wird eine Variablen-Liste an Argumenten übergeben. Diese werden durch `size_t n_args, size_t n_kw, const mp_obj_t *args` angegeben. Vergleichbar ist diese Art der Parameterübergabe mit dem 'argc' und 'argv' einer C++-Main-Methode. Die Besonderheit in MicroPython ist, dass die Angabe von Key-Word-Arugments unterstützt wird. Also zum Beispiel `sensor(port = 0)` wobei 'port' hier ein Key-Word-Argument ist.

Diese besonderen Methoden, welche durch das MicroPython-Klassen-Konzept bereitgestellt werden, werden in dem `MP_DEFINE_CONST_OBJ_TYPE`-Makro angegeben und somit an die MP-VM angebunden.

```
MP_DEFINE_CONST_OBJ_TYPE(
    mp_type_list,
    MP_QSTR_list,
    MP_TYPE_FLAG_ITER_IS_GETITER,
    make_new, mp_obj_list_make_new,
    print, list_print,
    unary_op, list_unary_op,
    binary_op, list_binary_op,
    subscr, list_subscr,
    iter, list_getiter,
    locals_dict, &list_locals_dict
    );
```

[MP, 'micropython/py/objlist.c' ab Zeile: 455]

> Important
>
> Eines der wohl wichtigsten Konzept hier ist das Anbinden eines 'locals_dict'. Dies ist konzeptionell dasselbe wie das Globals-Dictionary der Modul-Typen. Hier wird eine Dictionary für Funktions-Registrierungen übergeben.

Das Listen-Objekt verwendet eine Vielzahl dieser eingebauten Konzepte. An erster Stelle ist immer die `mp_obj_type_t` Referenz. An dieser Stelle bindet MicroPython den Typ-Pointer an die definierten Funktionen. In diesem Fall ist der Typ durch `mp_type_list` gegeben.

MP-Konzept	Bedeutung	Verwendung
MP_QSTR_`name`	Der Objekt Name	Dieser wird beispielsweise vom Python-Operator `type` verwendet, um den Namen eines Objekts auszulesen.
make_new	Der `new` Operator	Diese Funktion wird beim Erstellen einer neuen Objektinstanz aufgerufen. Es wird das MicroPython-Objekt als Rückgabewert wieder gegeben.
print	Die zum Objekt gehörende print Funktion	Das Binding für die Print-Funktion, `print(objekt)`.
unary_op	Unary Operation	Dies sind Operationen auf dem Objekt selber z.B. das Inverse eines Objektes.
binary_op	Binäre Operationen	Dies sind Operationen mit anderen Objekten z.B. der `+`- oder `-`-Operator.
locals_dict	Das Objekt-Dictionary	Im Objekt-Dictionary werden weitere Funktionen des Objektes festgehalten und definiert. Für Klasse-Typen sind diese instanzgebunden.
MP_TYPE_FLAG_ITER_IS_GETITER	Spezielles Iterier-Verfahren	Zuweisung der Iterator Implementation, also welche der verschiedenen MicroPython-Iterations-Verfahren verwendet werden soll.
subscr	Subscription	Dieses Konzept ist meistens als `Indexing` bekannt z.B. `a[1]`.
iter	Der Iterator	Dieses Objekt kann z.B. in einer For-Each-Loop verwendet werden.

Abbildung 9: MP-Konzepte

Funktions-Objekt-Definition

Wie in Klassen-Typ : Bereitgestellte Funktionen und Konzepte und Modul-Typ beschrieben, bieten Klassen so wie Module, die Möglichkeit ein Dictionary für Funktions-Registrierung anzubinden. Im Folgenden wird qualitativ erklärt, wie die hier anzubindenden Funktionen implementiert werden.

Wie bereits beschrieben werden Funktionen mithilfe eines QStrings an ihren Funktions-Namen gebunden. Dies ist Teil des Objekt-Dictionaries:

```
{ MP_ROM_QSTR(MP_QSTR_<FuntionsName>), MP_ROM_PTR(&<MicroPython-Funktions-Objekt>) },
```

Das MicroPython-Funktions-Objekt ist eine C-Funktion, welche zu einem MicroPython-Objekt konvertiert wurde. Für diesen Zweck werden verschiedene Makros angeboten.

Die Möglichkeiten MicroPython-Funktionen zu implementieren kann beliebig komplex werden. Daher wird die Erklärung dieses Prozesses auf das `MP_DEFINE_CONST_FUN_OBJ_1` Makro beschränkt. Dieses ermöglicht das Erstellen eines MicroPython-Objektes aus einer C-Funktion mit einem Argument. Im Wesentlichen lässt sich das Verwenden dieses Makros durch das folgende Muster darstellen:

```
static mp_obj_t <FuntionsName>(mp_obj_t <arg1>) {
    <...>
    return <MicroPython-Objekt>;
}
static MP_DEFINE_CONST_FUN_OBJ_1(<MicroPython-Funktions-Objekt>, <FuntionsName>);
```

Es wird eine C-Funktion definiert. Diese hat ein Argument. Das Argument der C-Funktion muss ein MicroPython-Objekt sein. Das hier übergebene Argument kann mit Hilfe des in MicroPython Typ Zuordnung vorgestellten Konzeptes auf einen Typ geprüft und verwendet werden. Der Rückgabewert einer in MicroPython eingebundenen Funktion ist immer ein MicroPython-Objekt, wobei Null auch ein valider Rückgabewert sein kann. Die durch MicroPython bereitgestellten Makros legen ein MicroPython-Funktions-Objekt an. Das MicroPython-Funktions-Objekt wird in ein Dictionary eingebunden, um dieses einem Modul oder einer Klasse zuzuordnen.

Im Laufe des Projektes sind verschiedene Variationen von Funktionen umgesetzt worden. Um einen Einblick in Argument Parsing

> **Note**
>
> Argument Parsing ist der Umgang mit Key-Word-Argumenten. Durch das Argument Parsing kann auf Key-Word-Argumente zugegriffen werden.

und Funktionen mit variablen Parameter zu erlangen, können die Modul-Dateien dieses Projektes betrachtet werden. Hier ist jedoch anzumerken das Prozesse der MicroPython-C-API durch das Einführen von weiteren Makros teilweise abstrahiert wurden.

MicroPython Typ Zuordnung

MicroPython stellt eine Möglichkeit bereit, um verschiede MicroPython-Objekt-Typen zu unterscheiden und unterschiedlich zu verarbeiten. Im Folgenden werden in diesem Zusammenhang ein paar Fragen geklärt:

1. Worauf stützt sich die Typ-Zuordnung
2. Wie funktioniert die Typ-Zuordnung (Nicht-Konkrete-Typen)
3. Wie funktioniert die Typ-Zuordnung (Konkrete-Typen)

Worauf stützt sich die Typ-Zuordnung

Wie in MicroPython-Object-Type beschrieben wird der `mp_obj_t` Pointer verwendet, um zusätzliche Informationen über das MicroPython-Objekt zu verwalten. Die für uns bereitgestellten Objekt-Klassifizierungen werden wie folgt unterschieden:

Typ	Bit
Small Int	xxxx...xxx1
QStrings	xxxx...x010
Intermediate Object	xxxx...x110
MP Object Base	xxxx...xx00

Abbildung 10: MicroPython-Typen

'Intermediate Objects' sind temporäre Objekte, die Daten während der Ausführung von C- und MicroPython-Code austauschen und umwandeln. Diese spielen bei der Implementierung eines eigenen MicroPython-Port keine große Rolle. Das Arbeiten mit diesen Objekten ist durch die MicroPython-C-API genauso wie mit jedem anderen Objekt. Intermedia-Objekte sind z.B. True, False, Strings und Integer. QStrings sind, wie in QStrings-Type beschrieben, eine spezielle Repräsentation von Strings. MP-Object-Base sind wie in MicroPython-Object-Type beschrieben, im Wesentlichen entweder Modul-Objekte, Klassen-Objekte, Funktionen oder Konstanten bzw. Variablen. Eben solche Objekte, welche eine MicroPython Objekt-Base besitzen.

Wichtig ist hier anzumerken, wie dieser Prozess der Speicherung von Typ-Informationen funktioniert.

MicroPython verfügt nicht über einen speziellen Mechanismus, um sicherzustellen, dass MP Object Base Pointer immer mit 'xxx...xx00' enden. Stattdessen verlässt sich MicroPython hier auf Pointer-Alignment. Diese Annahme basiert darauf, dass der GCC-Compiler verwendet wird. Da der GCC-Compiler dem C-Standard folgt, geht er immer von ausgerichteten Zeigern aus. Definiert wird dies in dem C-Standart:

"3.2 alignment requirement that objects of a particular type be located on storage boundaries withaddresses that are particular multiples of a byte address" [C-s, „3.2 alignment“ , Seite 17]

Da MicroPython bis zu 3 Bits verwendet, um Typ-Informationen zu speichern, nehmen die MicroPython-Entwickler hier an, dass die Kompilierung für das Zielsystem mindestens ein 3-Bit-Alignment aufweist.

Aus der ARM Compiler Toolchain Dokumentation ist zu entnehmen, dass alle Pointer 32-Bit-Aligned aufweisen. [vlg. ARM-al, „Size and alignment of basic data types“]

Somit ist die von den MicroPython-Entwicklern gemachte Annahme über das Alignment, für den von uns gewählten Mikrocontroller zulässig, ohne zusätzliche Änderungen im Compile-Prozess vorzunehmen.

Wie funktioniert die Typ-Zuordnung (Nicht-Konkrete-Typen)

Die in *"Abbildung 10: MicroPython-Typen"* angegebenen Daten-Typen sind im Sinne der Typ-Zuordnung ähnlich implementiert, daher hier 'Small Int' als Beispiel:

```
#if MICROPY_OBJ_REPR == MICROPY_OBJ_REPR_A

static inline bool mp_obj_is_small_int(mp_const_obj_t o) {
    return (((mp_int_t)(o)) & 1) != 0;
}
#define MP_OBJ_SMALL_INT_VALUE(o) (((mp_int_t)(o)) >> 1)
#define MP_OBJ_NEW_SMALL_INT(small_int) ((mp_obj_t)((((mp_uint_t)(small_int)) << 1) |
1))
```

[MP, 'micropython/py/obj.h', ab Zeile: 86]

Im Gegensatz zu `mp_obj_base_t` -Pointern wird hier der `mp_obj_t` -Pointer nicht als tatsächlicher Zeiger, sondern als Nutzdaten-Träger für den Wert des 'Small Int' verwendet. Außerdem als Informations-Träger für die Typ-Zuordnung. Dies funktioniert, da 'Small Int' weniger Bits benötigt als der Void-Pointer und somit Platz für die zusätzliche Information des Datentyps bietet. Hier ein Code-Schnipsel, welches die Funktionsweise verdeutlicht:

```
mp_obj_t b = MP_OBJ_NEW_SMALL_INT(0x1); //b soll den wert 1 haben
printf("Ausgabe(%d)", b); //Ausgabe(3) d.h. 0x..0011
                          //also 0x00000..x1 wir haben den typ small int
                          //mit dem Nutzdaten-Wert von (0x..011 > 1 =) 1
```

> **Note**
>
> Arbeitet man mit 'mp_obj_t' und erhält fälschlicherweise einen einfachen Datentyp als Parameter seiner Methoden, so ist es wichtig, folgendes zu beachten: Diese besonderen Datentypen werden anders behandelt. Anders als 'mp_obj_t', welches direkt auf ein Struct referenziert. Vor dem Casten auf ein tatsächlichhes MicroPython-Object solle man prüfen, ob das Eingabe-Objekt vom richtigen Typ ist. MicroPython bietet Funktionen für diese Typ-Prüfungen an. So kann man entweder eine der in die C-API eingebauten Funktionen zur Typ-Prüfung verwenden, wie z.B. oben beschrieben `mp_obj_is_small_int` , oder die MicroPython-Objekt-Base für den Typ-Vergleich verwenden. Mehr dazu im nächsten Abschnitt.

Wie funktioniert die Typ-Zuordnung (Konkrete-Typen)

Konrete-Typen sind wie bereits in Konkrete-Typen beschrieben eben solche Typen, welche über eine Objekt-Base verfügen. Und damit auch über eine `mp_obj_type_t` also einen MicroPython-Objekt-Typ.

```
// Anything that wants to be a concrete MicroPython object must have mp_obj_base_t
// as its first member (small ints, qstr objs and inline floats are not concrete).
struct _mp_obj_base_t {
    const mp_obj_type_t *type MICROPY_OBJ_BASE_ALIGNMENT;
};
typedef struct _mp_obj_base_t mp_obj_base_t;
```

[MP, 'micropython/py/obj.h', ab Zeile: 52]:

In der Typ-Tabelle ist unter Worauf stützt sich die Typ-Zuordnung beschrieben, dass 'MP-Object-Base' eine `mp_obj_t`-Adresse ist. Die letzten Bits dieser Addresse sind auf 'xx...xx00' gesetzt. Dies bedeutet, dass es sich hier um eine durch das Pointer-Alignment entstandene Variable handelt. Wird dieses Muster bei einem MicroPython-Objekt erkannt, so kann davon ausgegangen werden, dass es sich um einen Konkreten-Typ handelt. Auf diese Annahme gestützt kann der `mp_obj_t`-Pointer zu einem `mp_obj_base_t`- oder `mp_obj_type_t`-Pointer gecastet werden. Wenn das MicroPython-Objekt korrekt konfiguriert wurde, ist nun der MicroPython-Objekt-Typ auslesbar. Und das MicroPython-Objekt kann einem Typ zugeordnet werden.

Eben diese Operation wird durch die getType-Operation bzw. desser Wrapper-Funktion realisiert:

```
const mp_obj_type_t *MICROPY_WRAP_MP_OBJ_GET_TYPE(mp_obj_get_type)(mp_const_obj_t o_in)
{
#if MICROPY_OBJ_IMMEDIATE_OBJS && MICROPY_OBJ_REPR == MICROPY_OBJ_REPR_A
    if (mp_obj_is_obj(o_in)) {
        const mp_obj_base_t *o = MP_OBJ_TO_PTR(o_in);
        return o->type;
    } else {
<...>
```

[MP, 'micropython/py/obj.c', ab Zeile: 56]

Der Typ eines MicroPython-Objektes ist durch einen Typ-Pointer von diesem definiert. Jede MicroPython-Objekt-Datei definiert diesen für sich. Meist sieht dies wie folgt aus:

```
//Deffinieren des Typ-Pointers
const mp_obj_type_t motor_type;
//Einstellen des Objekt-Typen
motor_obj_t m = { .base = { .type = &motor_type }, <Weitere_Objekt_Felder> };
```

Es wird ein Konstanter-Typ-Pointer erstellt. Es wird die Einmaligkeit einer Speicher-Adresse verwendet, um jedem Objekt-Typen eine eindeutige Zuordnung zu einem Typ zuzuweisen.

Ein solcher Typ-Vergleich könnte wie folgt aussehen:

```
//Die oben gezeigte MICROPY_WRAP_MP_OBJ_GET_TYPE-Funktion wird intern verwendet.
mp_obj_get_type(motor) == &motor_type
```

Bei Konkrete-Typen ist also eher wichtig das MicroPython-Objekte korrekt aufgebaut sind. Wie ein korrekt aufgebautes MicroPython-Objekt aussieht, wird im weiteren erklärt. Unter Berücksichtigung der folgenden Einschränkungen wird eine Objekt-Typ-Zuordung zu einem einfachen Cast und Zugriff auf das Feld einer Struktur.

Durch die Natur der 'getType'-Funktion, ist das Zugreifen auf einen Objekt-Typ ein riskanter Cast. Es funktioniert jedoch aufgrund des gut durchdachten Speicher- und Objekt-Layouts der MicroPython-Entwickler. So wird an dieser Stelle vorausgesetzt das 'mp_obj_t'-Pointer auf Objekte zeigen, welche an der Adresse 0 ihrer Struktur eine 'mp_obj_base_t' haben. Sonst funktioniert dieser Cast nicht. Dies ist eine Besonderheit der 'C-Structs' die von den MicroPython-Entwicklern genutzt wird.

Angenommen wir haben diese 'Struct':

```
typedef struct _button_obj_t {
    mp_obj_base_t base;
    uint8_t id;
} button_obj_t;
```

Dann wird an der ersten Stelle des 'Struct', also an Adresse 0, die 'mp_orb_base_t' liegen, solange dies nicht durch Optimierung verändert wurde. Auf dieses folgt, inklusive Alignment, die aneinander gereihten anderen 'Struct'-Variablen. Der Cast funktioniert.

Betrachten wir nun dieses 'Struct':

```
typedef struct _button_obj_t {
   uint8_t id;
   mp_obj_base_t base;
} button_obj_t;
```

Tauschen wir die Reihenfolge der Parameter, so produziert der Typ-Zugriff einen Speicherzugriffsfehler, da `const mp_obj_base_t *o = MP_OBJ_TO_PTR(o_in)` an dieser Stelle auf das `uint8_t id` zeigt. Wird diese Besonderheit von MicroPython-Objekten bei der Implementierung beachtet, so ist die Typ-Prüfung ein sicherer Vorgang.

Problematik bei der Verwendung von Namespaces

Bei dem Versuch MicroPython-Module und Klassen aufzubauen ist ein Problem aufgefallen. Es gibt Limitierungen bei der Verwendung von Namespaces. Auf diese und die daraus entstehende Problematik soll im Folgenden eingegangen werden.

- Super- & Submodule
- Limitierungen des MicroPython-Interpreters
- Submodule als QString Alias
- Kombination von Submodulen, QString und Supermodulen
- Schlussfolgerung zu diesen Problemen

Super- & Submodule

Dies ist eine kurze Begriffs-Erklärung. Super-Module sind solche Module, welche andere Module als Teil ihres Dictionaries verwalten. Also solche Module, über die man auf andere Module zugreifen kann. Ein Submodul ist ein solches Modul, welches von einem Supermodul eingebunden wurde.

Limitierungen des MicroPython-Interpreters

Imports von Modulen funktionieren nur eine Ebene tiefer. Hat man z.B. ein Modul `Devices` und registriert darin das Submodul `Sensors`, welches wiederum einen spezifischen Sensor wie `EV3-Lichtsensor` enthält, so könnte man erwarten, dass die Syntax `import Devices.Sensors.EV3-Lichtsensor` oder `from Devices.Sensors import EV3-Lichtsensor` funktioniert. Jedoch findet der MicroPython-Interpreter diese Module nicht.

Für solche Submodule bietet MicroPython ein alternatives Vorgehen an. Das Registrieren von 'Submodule als QString Alias'. Dieses wird im Subpackage-Beispiel des MicroPython-Projektes erklärt (siehe [MP, 'micropython/examples/usercmodule/subpackage']). Diese Methode bringt jedoch andere Probleme mit sich.

Submodule als QString Alias

Ein alternatives Vorgehen für solche Submodul-Strukturen wäre, das Modul `Devices` gar nicht umzusetzen.

Man würde ein Modul `Sensor` schreiben, registriert dieses jedoch nicht bei einem Supermodul, sondern gibt ihm den Namen `MP_QSTR_devices_dot_sensors`. Anschließend muss man eine QString-Datei erstellen. Diese wird in den MicroPython-Build-Prozess eingebunden. In dieser werden die 'dot-strings' ein weiteres Mal definiert, `Q(devices.sensors)`. Dieser Schritt ist notwendig um "dot" durch "." für den Import-Namen zu ersetzen.

Die damit verbundenen Probleme sind folgende: Das Devices-Modul kann keine eigenen Funktionen verwalten. Da es praktisch gesehen gar nicht existiert. Außerdem kann das Devices-Modul selber nicht importiert werden. Die Anweisung `import devices` führt zu einem Import-Error. Es muss stattdessen `from devices.sensors import EV3-Lichtsensor` für den Import verwendet werden.

Kombination von Submodulen, QString und Supermodulen

Man könnte auf die Idee kommen, beide Ansätze zu kombinieren. Man erstellt ein Supermodul `devices`, registriert dieses sowie das zugehörige Submodule. Das Submodule selbst wird ebenfalls registriert. Dieses Vorgehen scheint auf den ersten Blick zu funktionieren und löst die in den beiden vorangegangenen Abschnitten beschriebenen Probleme, hat jedoch auch unerwartetes Verhalten.

Führt man `import devices.sensors` aus, so scheint der Import korrekt zu funktionieren. Bei näherer Betrachtung wird jedoch nicht das Modul `sensors` importiert. Es wird hier das Modul `sensors` unter dem Namen `devices` importiert. Führt man dann `print(devices)` aus, so wird der Name des `sensors` Moduls ausgegeben. Das Devices-Modul wurde hier überschrieben. Erst durch den Befehlt `import devices` ist dieses wieder korrekt erreichbar.

Schlussfolgerung zu diesen Problemen

Ursprünglich war geplant, den Namespace `orb` zu verwenden, um Module zu organisieren. So könnte man Klassen, Module und Funktionen unter einem Namespace zusammenfassen. Wie z.B. `orb.sensors` oder `orb.devices`. Aufgrund der oben genannten Probleme ist dies jedoch nicht möglich. Der `orb` -Prefix muss gestrichen werden, um das von MicroPython erwartete Verhalten zu gewährleisten.

Es gibt daher die Einschränkung, Module nur zwei Ebene tief zu gestalten. Funktionen und Klassen auf einer dritten Ebene werden nur registriert, wenn sie ausschließlich im Zusammenhang mit dem Supermodul Sinn ergeben. Wie zum Beispiel Funktionen einer Klasse, die zuvor instanziiert werden muss.

In der Praxis ist dies zum Beispiel `devices.sensor.get()`. Die Get-Funktion eines Sensors ergibt nur Sinn, falls sie im Zusammenhang mit einem Sensor-Objekt verwendet wird.

MicroPython Flags

Die MicroPython Flags werden in der Datei `mpconfigport.h` definiert. Ein Auszug aus dieser Datei könnte so aussehen:

```
#define MICROPY_CONFIG_ROM_LEVEL                (MICROPY_CONFIG_ROM_LEVEL_MINIMUM)

#define MICROPY_PERSISTENT_CODE_LOAD            (1)
#define MICROPY_ENABLE_COMPILER                 (1)

#define MICROPY_ENABLE_GC                       (1)
#define MICROPY_PY_GC                           (1)
#define MICROPY_FLOAT_IMPL                      (MICROPY_FLOAT_IMPL_FLOAT)
```

Diese Flags werden benutzt um MicroPython mitzuteilen, welche Module zu dem Port hinzugeladen werden sollen und welche nicht. Es gibt eine ganze Menge an Standard-Modulen, welche für dieses Projekt nicht gebraucht werden. Daher ist als erstes die Verwendung von `#define MICROPY_CONFIG_ROM_LEVEL (MICROPY_CONFIG_ROM_LEVEL_MINIMUM)` hervorzuheben. Dies bewirkt, dass keine Standard-Module verwendet werden. Es wurde also das absolut minimale Grundgerüst der MP-VM konfiguriert. Darauf folgen zwei weitere wichtige Flags. `MICROPY_PERSISTENT_CODE_LOAD` fügt den MicroPython-Byte-Code-Interpreter hinzu. Dieser soll am Ende dieses Projektes der einzige Weg sein, MicroPython-Programme auszuführen. `MICROPY_ENABLE_COMPILER` hingegen in der String Interpreter. Dieser war zu Beginn des Projektes noch hinzugeschaltet. Er konnte für Debug-Zwecke verwendet werden. Im fertigen Projekt würde diese Flag jedoch de-aktiv sein.

Reduzierung des MicroPython-Heap-Verbrauchs durch Objektreferenzen

MicroPython hat einen sehr begrenzten Heap-Speicher. Dies stellt einige Probleme dar, die bei der Implementierung einer MicroPython-Firmware sowie beim Arbeiten mit MicroPython selbst berücksichtigt werden müssen. Ein Problem, das in diesem Zusammenhang identifiziert wurde, wird im Folgenden erläutert. Dabei wird dies beispielhaft für einen Daten-Typ erklärt. Dieses Problem tritt jedoch auch bei anderen Daten-Typen auf.

Betrachteten wir als Beispiel den Float-Daten-Typ. In MicroPython sind Floats MicroPython-Objekte. Diese bilden das Verhalten, welches man von Floats erwartet, ab. Sie sind ein Konkretes-MicroPython-Objekt. Nehmen wir an, wir haben den folgenden Code:

```
import gc
a = 0.5
while a < 10000:
    a = a + 0.5
    print(gc.mem_free())
```

Jedes Mal, wenn `a = a + 0.5` aufgerufen wird, liest die MP-VM den alten Wert von a und erstellt ein neues Objekt mit dem neuen Wert. Das alte Objekt bleibt jedoch im Speicher erhalten. Dieses kleine Programm kann sehr schnell zu einem Speichermangel führen. Grund dafür ist unzureichende Heap-Speicherkapazität. Wir können diesen Überlauf verfolgen, indem wir den freien Speicher mit `print(gc.mem_free())` ausgeben. Eine Lösung für dieses Problem besteht darin, den integrierten Garbage-Collector zu verwenden.

> **Note**
>
> Dies muss nicht zwangsläufig, kann aber zu einem Fehler in der Programm-Ausführung führen. Das Beispiel oben erlaubt der MP-VM zu erkennen, dass nicht genügend Heap-Speicher zur Verfügung steht. In diesem Fall wird der 'gc.collect()'-Befehl automatisch durch die MP-VM durchgeführt. Gegebenenfalls wird die MP-VM Ausführung unterbochen. Unter bestimmten Bedingungen kann dies jedoch zu einem Speicher-Zugriffs-Fehler führen. Dieser Fehler wird in Tests,Spezifikationen.md unter 2.2.4. Error-Tests: 'Hard-Fault' provoziert.

```
import gc
a = 0.5
while a < 10000:
    a = a + 0.5
    gc.collect()
    print(gc.mem_free())
```

Wenn man nun die Ausgabe von `gc.mem_free()` betrachten, sieht man, dass der verwendete Speicher nicht unbegrenzt ansteigt. Dies ist eine gute Lösung, aber der Nutzer muss über dieses Problem Bescheid wissen. Und in diesem Zusammenhang über die möglichen Risiken.

Eine mögliche Lösung für dieses Problem ist das Integrieren eines Aufrufs zum Garbage Collector in die Routine zur Erstellung neuer Objekte. Also die gleichen Überprüfungen, welche MicroPython bereits für eigene Daten-Typen integriert hat. Dies könnte eine Lösung sein, wenn das Hinzufügen von möglicherweise überflüssigen Overhead kein Problem darstellt. In den meisten Fällen hat man bei Mikrocontrollern jedoch eine starke Performance-Begrenzung. Daher ist dies wohl zumeist keine gute Lösung. In diesem Projekt sind am stärksten die Klassen für Sensoren, Motoren usw. betroffen. Eben all diese Klassen, welche ein an einen Port angeschlossenes Gerät abbilden. Daher wurde folgende Lösung gewählt. Es wurde eine Liste mit einem Objekt für jeden Port erstellt. Hier für die zwei Servo-Ports:

```
servo_obj_t servo_obj_list[2] = {
    { .base = { .type = &servo_type }, .port = 0, .speed = 0, .angle = 0 },
```

```
    { .base = { .type = &servo_type }, .port = 1, .speed = 0, .angle = 0 },
};
```

Wann immer ein Nutzer ein neues Objekt erstellt, erhält er eine Referenz auf eines dieser statisch erstellten Objekte. Betrachten wir nun den folgenden MicroPython-Code:

```
from devices import servo
a = servo(0)
b = servo(0)
a.set(speed=10, angle=20)
```

Dieser Code-Schnipsel hat folgenden Nebeneffekt. Es wird nicht nur der Zustand von Servo-`a` aktualisiert, sondern auch Servo-`b` beeinflusst, da diese beiden ein gemeinsames Objekt verwalten. Auf diese Weise treten keine Speicherprobleme im Zusammenhang mit dem zuvor erwähnten Problem auf. Zusätzlich hat der Nutzer immer Objekte, die den aktuellen Zustand der realen Geräte oder zumindest der angewendeten Einstellungen repräsentieren.

Betrachten wir nun diesen MicroPython-Code:

```
a = servo(0)
b = servo(0)
a.set(speed=20)
b.set(angle=30)
```

Der Servo wird auf `angle(30)` mit `speed(20)` bewegt. Es wird also zusammenfassend ein konsistentes Verhalten zwischen den verschiedenen Devices-Objekten gewährleistet und der MicroPython-Heap-Verbrauch minimiert.

Thread Safety

MicroPython ist nicht Thread-Safe. Dies ist bei der Implementierung der MicroPython-Task zu beachten. MicroPython selber sollte, solange es nicht anders möglich ist, seinen Speicher-Bereich vollständig selber verwalten. Die MicroPython-Task sollte vollständig losgelöst von der restlichen Firmware laufen. Kommunikation zwischen ORB-Firmware und Python-VM sollte nur über Flags geschehen, die immer nur von einer der beiden Seiten beschrieben werden können. Dadurch entstehende Race-Conditions sollten bedacht werden.

> MicroPython selber bietet die Möglichkeit an, Threads zu verwalten. Dies jedoch nur für ein Thread-Modul welches in der MP-VM in sich geschlossen arbeitet. Dafür muss man die in der Datei `mpthread.h` definierten Funktionen umsetzen. Da die ORB-Application von sich aus keine Threads anbietet und das Umsetzen dieser somit außerhalb des Umfangs dieser Bachelorarbeit liegt, wird dies nicht weiter beachtet.

Windows-Bug: Falsches Register bei Non-Local Return-Adressierung

- Definition und Funktionsweise des Non-Local Return
- Beschreibung des Bugs
- Vorgehensweise zur Fehlerbehebung

Definition und Funktionsweise des Non-Local Return

Der Non-Local Return (NLR) wird in der MicroPython-Umgebung verwendet, um bei Exceptions oder Fehlern schnell aus verschachtelten Funktionsaufrufen herauszuspringen. Dabei ermöglicht der NLR es, den normalen Programmablauf zu unterbrechen und direkt zu einem vorher definierten Punkt im Code zurückzukehren. Und dies ohne erst den vollständigen Funktionsaufruf nach oben hin wieder abzuwickeln. Z.B. bei dem Auftreten von Exceptions. Hier muss der Programmablauf effektiv und vor allem speicher- und rechenleistungs-effizient unterbrochen werden können.

Beschreibung des Bugs

Führt man den MicroPython-Embed-Port unter Windows aus, ohne die Optimierungs-Flag `-fomit-frame-pointer` zu setzen, so kommt es zu einem Speicherzugriffsfehler nach der Ausführung der Funktion `nlr_push` . Diese Funktion ist in der Datei `micropython/py/nlrx64.c` zu finden. Diese Option ist in `-Os` & `-Og` enthalten, wie in der GNU-GCC-Dokumentation beschrieben ist. [vgl. GCC-cf] Die `-fomit-frame-pointer` -Option ist also in den beiden als sicher angesehenen Optimierungs-Flages für dieses Projekt enthalten. Mehr zu den Optimierungs-Flages unter Compiler Flag Kompatibilität. "-fomit-frame-pointer <...> Enabled by default at -O1 and higher." [GCC-cf, „-fomit-frame-pointer“]

> **Note**
>
> Da `-Os` alle Optimierungen von `-O1` durchführt. Und `-Os` alle von `-O2` . (Beide Optimierungen haben Ausnahmen, zu denen `-fomit-frame-pointer` jedoch in beiden Fällen nicht zählt).

Der Grund für diesen Fehler ist, die Annahme, dass MicroPython an folgender Stelle die `omit-frame-pointer` -Anweisung voraussetzt.

```
#if !MICROPY_NLR_OS_WINDOWS
#if defined(__clang__) || (defined(__GNUC__) && __GNUC__ >= 8)
#define USE_NAKED 1
#else
// On older gcc the equivalent here is to force omit-frame-pointer
__attribute__((optimize("omit-frame-pointer")))
#endif
//Hier müsste Windows eigentlich sein eigenes Handling für omit-frame-pointer haben
#endif
```

[MP, 'micropython/py/nlrx64.c', ab Zeile 38]

Ungünstig in an dieser Stelle nur, dass der Fall für Windows hier gar nicht beachtet wird. Das Windows diese Optimisierungs-Anweisung jedoch erwartet, auch wenn nicht hier angeführt, kann man aus der Funktion jedoch herauslesen.

Es handelt sich hier um ein Problem, welches im Rahmen des 'nlr_push'-Assembler-Codes auftritt. Daher ein wenig Kontext zu diesem Problem. Bei diesem Problem spielen zwei Register eine Rolle, 'rbp' und 'rsp'. Das 'rsp'-Register ist das Stack Pointer Register. Das 'rbp'-Register is das Stack-Frame Base Pointer Register. [vgl. AMD, „2.4 Stack Operation“]

Wenn der Compiler 'rbp' als Frame Pointer verwendet,

> **Note**
>
> Dies könnte man als non-omitted bezeichnen.

sorgt er dafür, dass 'rbp' auf den Anfang des Stack-Rahmens zeigt. [vgl. AMD, „2.4 Stack Operation“] Ein Stack-Rahmen speichert die lokalen Variablen, Funktionsargumente und die Rücksprungadresse einer Funktion. Ist 'rbp' als Frame-Pointer verwendet, so kann man diese Register nicht mehr frei verwenden. Es ist kein General-Purpose-Register mehr.

> "-fomit-frame-pointer Omit the frame pointer in functions that don’t need one. This avoids the instructions to save, set up and restore the frame pointer; on many targets it also makes an extra register available." [GCC-cf]

Der Quell-Code der betroffenen Funktion sieht wie folgt aus:

```
unsigned int nlr_push(nlr_buf_t *nlr) {
    #if !USE_NAKED
    (void)nlr;
    #endif

    #if MICROPY_NLR_OS_WINDOWS

    __asm volatile (
        "movq   (%rsp), %rax        \n" // load return %rip
        "movq   %rax, 16(%rcx)      \n" // store %rip into nlr_buf
        "movq   %rbp, 24(%rcx)      \n" // store %rbp into nlr_buf
        "movq   %rsp, 32(%rcx)      \n" // store %rsp into nlr_buf
        "movq   %rbx, 40(%rcx)      \n" // store %rbx into nlr_buf
        "movq   %r12, 48(%rcx)      \n" // store %r12 into nlr_buf
        "movq   %r13, 56(%rcx)      \n" // store %r13 into nlr_buf
        "movq   %r14, 64(%rcx)      \n" // store %r14 into nlr_buf
        "movq   %r15, 72(%rcx)      \n" // store %r15 into nlr_buf
        "movq   %rdi, 80(%rcx)      \n" // store %rdr into nlr_buf
        "movq   %rsi, 88(%rcx)      \n" // store %rsi into nlr_buf
        "jmp    nlr_push_tail       \n" // do the rest in C
        );

    #else
```

[MP, 'micropython/py/nlrx64.c', ab Zeile: 59]

Hier ist an folgender Zeile zu erkennen, was falsch läuft.

```
`movq   (%rsp), %rax        \n" // load return %rip`
```

Wie auch im Kommentar steht, wird hier die Adresse für den return Sprung geschrieben. Diese jedoch nicht nach 'rbp', sondern nach 'rsp'.

Wird 'rbp' 'omitted', so wird das 'rsp'-Register direkt verwendet, um das Stack-Top zu verwalten. Dies gibt vor, wo sich die Rücksprung-Adresse befindet.

Das heißt, hier ist es notwendig, dass das 'rsp'-Register und nicht das 'rbp'-Register von dem Compiler als "Rücksprung-Zuständiger" erkannt wird. Falls nicht, ist das Verhalten undefiniert und wir bekommen im besten Fall einen Speicher-Zugriffs-Fehler, können somit erkennen, dass hier etwas schiefgelaufen ist.

Vorgehensweise zur Fehlerbehebung

Da dieses Problem nur unter Windows eine Rolle spielt und für den STM32F405 Mikrocontroller dies Problem nicht auftritt, reicht es mit Compiler-Flags zu Kompilieren, welche `-fomit-frame-pointer` setzen. Also `-Os` oder auch `-Og`. Das ORB verwendet in seinem Kompilier-Prozess `-O0`, jedoch gibt es diesen Bug nicht bei der Verwendung mit diesem Mikrocontroller. Das bedeutet, das Omit-Frame-Pointer-Attributes wird korrekt gesetzt, da die für diesen Mikrocontroller zuständigen Dateien dies korrekt implementieren. Im Verlaufe des Projektes wurde die `nlrx64.c` -Datei um Logik für das Setzen des Omit-Frame-Pointer-Attributes erweitert. Es muss nun nicht mehr das Flag `-fomit-frame-pointer` für den gesammten Build-Prozess gesetzt werden. Dies bedeutet auch, das Windows-Projekt kann mit `-O0` genauso wie das Firmware-Projekt, kompiliert werden.

Compiler Flag Kompatibilität

> Caution
>
> Compiler-Flags sind ein Stolperstein für alle, die einen eigenen MicroPython-Port aufbauen wollen oder wie in diesem Projekt, in ein bereit bestehendes Projekt integrieren wollen. Ist man bei seinem Ausgangsprojekt, wie in diesem Fall, an bestimmte Compiler-Flags gebunden, so kann es grundsätzlich unmöglich oder sehr umständlich sein MicroPython in das gewählte Projekt zu integrieren. Die Betrachtung der kompatiblen Compiler-Flags sollte also vor der Implementierung des Ports getestet und gut durchdacht werden.

- MicroPython Compiler Flags
- ORB-Firmware Compiler Flags
- Angepasste Compiler Flags

MicroPython Compiler Flags

Möchte man MicroPython kompilieren und auch debuggen können, so ist es notwendig '-Og' als Flag zu setzen. Dieses Flag sorgt für eine moderate Optimierung. Hier ist an dieser Stelle aber wichtig, dass die Debug-Informationen erhalten bleiben. Durch die Verwendung dieser Flag ist es möglich in der Code::Blocks Umgebung MicroPython-Code sinnvoll zu debuggen.

> Die Verwendung dieser Flag erfüllt einen zweiten Zweck. Der Dokumentation der GNU-GCC-Compiler-Dokumentation ist zu entnehmen, dass viele Optimierung-Flags durch das Verwenden von '-Og' unterdrückt werden. Wie man im Kapitel „3.11 Options That Control Optimization“ nachlesen kann: "Most optimizations are completely disabled at -O0 or if an -O level is not set on the command line, even if individual optimization flags are specified. Similarly, -Og suppresses many optimization passes." [GCC-cf, „3.11 Options That Control Optimization“]

Auch wenn es noch viele weitere gibt, hier exemplarisch die `-ftree-pta` -Flag:

> ```
> -ftree-pta
>    Perform function-local points-to analysis on trees.
>    This flag is enabled by default at -O1 and higher, except for -Og.
> ```

[GCC-cf, „-ftree-pta“]

Dieser Ansatz minimiert also das Risiko von Problemen, welche durch aggressivere Optimierungen provoziert werden können. Das dies auch wirklich so ist, lässt sich einfach ausprobieren. Testweise wurde die `-Og` Flag entfernt und die `-O3` Optimierung verwendet. Dies ist eine sehr aggressive Optimierung. Wird der MicroPython-Port der mit der richtigen Compiler Flag problemlos funktioniert gebaut, so bekommt man schon vor dem Ausführen der ersten Python-Zeile den Fehler-Code `0xC0000005` . Dies ist ein Speicherzugriffs-Fehler.

> Note
>
> Im aktuellen Stand dieses Projektes lässt sich dieser Fehler nicht mehr produzieren. Es ist jedoch möglich, diesen durch das Verwenden des Embed-Example-Port zu produzieren.

Die Natur von MicroPython basiert stark auf dem Casten von Zeigern und dem dynamischen Aufrufen von Funktionen und das durch diese Operationen realisierte Aufrufen von Python-Funktionen. Es ist entscheidend, dass Optimierungen, die beispielsweise ungenutzte MicroPython-Module entfernen würden, nicht genutzt werden. Daher sollte im besten Fall auf aggressive Optimierungen verzichtet werden, um die Funktionsfähigkeit und Stabilität des Systems zu gewährleisten.

Ist das Debuggen nicht mehr notwendig, so ist es möglich `-Os` zu verwenden, um Speicherplatz zu sparen. Abgesehen von der geringeren Programm-Größe, bietet dieses Flag jedoch keinen Vorteil. Wichtig anzumerken ist, dass dies die einzige aggressive Optimierung-Flag ist, welche hier als sicher im Rahmen der Verwendung mit MicroPython angesehen wird. Diese lässt sich in einer Vielzahl der MicroPython-Ports finden.

ORB-Firmware Compiler Flags

Die ORB-Firmware hat als vor-konfigurierte Optimierung `-O0` eingestellt. Zusätzlich sind die Flags: `-ffunction-sections` und `-fdata-sections` verwendet. Diese Optionen isolieren Daten und Funktionen jeweils in ein eigenes Segment. Aus der GNU-GCC-Compiler-Dokumentation ist zu entnehmen, dass diese Flags nur den generierten Code sortieren. Jedoch nicht ungenutzten Code entfernen. Für das Entfernen von ungenutztem Code, hier "garbage collection", müsste man die Flag `-Wl,--gc-sections` setzen. [vgl. GCC-co] Daher ist diese Flags als eher unproblematisch einzustufen. Zusätzliches Testen konnte validieren, dass diese Compiler-Flags mit dem MicroPython-Projekt kompatibel sind. Außerdem ist in dem ORB-Firmware-Projekt konfiguriert, dass Floating-Pointer Operationen als**"FPU-specific calling convention"**generiert werden.

> **Note**
>
> FPU steht für Floating-Point-Unit. Die Microcontroller-Komponente, welche für Float-Operationen zuständig ist.

Dies bedeutet, dass hier Float-Operationen als Daten-Typ verwendet werden können. Dabei muss man sich keine Gedanken über die genaue Verwendung von Floats auf der FPU des Mikrocontrollers machen. Alle anderen Flags sind für Compiler-Warnings und zusätzliche Debug-Informationen.

Angepasste Compiler Flags

Da die Compiler Flags der ORB-Firmware ohnehin schon kompatibel mit den erlaubten Compiler-Flags des MicroPython-Ports sind, müssen keine weiteren Änderungen vorgenommen werden. Das fertige Projekt soll mit `-O0` und den zusätzlichen Flags der ORB-Firmware kompiliert werden.

Wird ein höherer Grad an Optimierung vonnöten sein, zum Beispiel durch unzureichenden Flash-Speicherplatz, so sollte im besten Fall die Optimierungs-Stufe `-Os` verwendet werden. Bei Verwendung dieser Flag sollte deren Kompatibilität im Zusammenhang mit der ORB-Firmware getestet werden. Zu diesem Zeitpunkt ist jedoch keine zusätzliche Optimierung notwendig. Dies könnte jedoch bei zusätzlicher Erweiterung der ORB-Firmware durch zum Beispiel zusätzliche Funktionen in der Zukunft erforderlich sein.

QStrings

QStrings sind Strings, welche gehashed wurden. Sie werden über diesen Hash-Wert verwaltet und zum Beispiel auf Gleichheit geprüft. Im Wesentlichen sind QString nur eine Zahl, die einen String repräsentieren. Die MicroPython-C-Api hat dabei interne Vorgänge, welche Kollisionen handhaben. Es gibt zwei Arten von QStrings. Solche, die als "dynamisch" bezeichnet werden können. Diese werden zu Laufzeit berechnet. Außerdem gibt es solche, die im MicroPython-Port-Pre-Compile-Schritt zugeordnet werden. Letzteres wird für Modul-Meta-Informationen, Klassen-Meta-Informationen oder allgemein für MicroPython-Objekt-Meta-Informationen verwendet. Diese sind an dem Präfix 'MP_QSTR' erkennbar. Durch dieses Vorgehen muss ihr Wert nicht zu Laufzeit berechnet werden. Wird jedoch eine QString dieser Form verändert, so muss der MicroPython-Port erneut gebaut werden. In diesem Projekt wird dieser Schritt auch als 'Rebuild-MicroPython' bezeichnet.

Test und Spezifikationen

Inhaltsverzeichnis

1. Testen in der Code::Blocks Umgebung

Im Ordner `"<PATH_TO_ORB>ORB-Python\Program"` befindet sich eine Datei 'program.py'. Diese kann verwendet werden, um Funktionalitäten der Micropython-VM zu testen. Diese Datei wird von der ORB-Windows-Application geladen und ausgeführt. Hier kann man, wenn man von der Code::Blocks-Umgebung aus testen möchte, sein Programm umschreiben.

Der Python-Api-Dokumentation (sphinx-docs) ist zu entnehmen, welche Funktionen bereit gestellt werden. Die erwarteten Ausgaben der Funktionen sind den Mockup-Dateien zu entnehmen. Grundsätzlich reicht hier einfaches Testen. Für diesen Zweck wurde ein Test-Ordner erstellt: `<Path_To_ORB>/ORB-Python-Exampes/Tests/Windows`.

Die Datei `_execute_(drag_test_here).bat` ist ein kleines Hilfs-Script um Tests einfach ausführen zu können. Hier ist wichtig, dass das Code::Blocks-Projekteinmal gebaut werden muss.

1.1. Windows Test-Spezifikation

- Es sollen alle Funktionen der Python-Api getested werden. Es sollen alle Funktionen einmal ausgeführt werden. Der Test-Vorgang ist wie folgt:
  - Jedes Modul bekommt einen eigenen Test
  - Funktionen mit Rückgabe-Wert: der Befehl print(`<funktions aufruf>`) ist zu verwenden
  - Funktionen mit Liste als Rückgabe-Wert: Hier wird zusätzlich auf die Listen-Elemente zugegriffen
  - Es soll die korrekte Verwendung von Objekt-Referenzen, wie z.b. bei Motoren, überprüft werden
  - Es sollen die Funktionen der Python-VM getestet werden
    - Exception-Handling und Exception-Exit-Status
    - Interrupt-Programm und Interrupt-Exit-Status
    - Der Normal-Exit-Status ist implizit in den anderen Tests vorhanden

1.2. Windows Tests

test.compilationError	Dieser Test ist für das Compile-Script. Er sollte einen Value-Error werfen. Zusätzlich kann man versuchen dieses Script als das `program.py` in Code::Blocks zu verwenden. Kompiliert man die ORB-Firmware sollte der Build-Prozess mit entsprechendem Error abgebrochen werden.

test.exception	Dieser Fehler überprüft ob MicroPython-Exceptions als solche erkannt werden (Exit-Code 1). Zusätzlich sollte die Art der Exception ausgegeben werden. In diesem Fall `Exception`.
test.memFault	Dieses Test-Script überprüft Memory-Allocation-Errors d.h. ungenügender Speicher im Micropython-Heap. Die VM soll mit dem Exit-Code Exception(1) beendet werden. Der Exit-Typ sollte einen Memory-Error anzeigen.
test.memory	Ein einfacher Test der Memory Funktionen. Es sollen alle Funktionen des Memory-Moduls verwendet werden.
test.monitor	Ein einfacher Test der Monitor Funktionen. Es sollen alle Funktionen des Monitor-Moduls verwendet werden.
test.motor	Ein einfacher Test der Motor Funktionen. Es sollen alle Funktionen des Motor-Moduls verwendet werden.
test.sensor	Ein einfacher Test der Sensor Funktionen. Es sollen alle Funktionen des Sensor-Moduls verwendet werden.
test.servo	Ein einfacher Test der Servo Funktionen. Es sollen alle Funktionen des Servo-Moduls verwendet werden.
test.time	Ein einfacher Test der Time Funktionen. Es sollen alle Funktionen des Time-Moduls verwendet werden.
test.userInt	Testen des User-Interrupts. Der Exit-Status sollte "2" sein und aus dem Exit-Typen sollte hervor gehen, dass ein User-Interrupt verwendet wurde.

Abbildung 11: Windows Tests

Alle Windows-Tests wurden erfolgreich durchgeführt. Das durch die Tests vorgegebene Verhalten konnte validiert werden. Die Windows-Umgebung erfüllt alle Anforderungen, die an sie gestellt wurden.

2. Testen der ORB-Firmware

Alle Firmware Tests wurden mit einem ORB, der HW-Version 0.22 durchgeführt. Es wurde ein Test-Programm für die ORB-Firmware erstellt. Dieses befindet sich unter 'ORB-Python-Examples/Tests/Firmware/firmware_test.py'. Das Test-Programm kann mit Hilfe des ORB-Monitors übertragen und ausgeführt werden. Dazu wird der Befehl `orb-util -f firmware_test.py` verwendet. Das Test-Programm bietet eine interaktive Oberfläche um alle Funktionen des Python-Interpreters zu testen. Das Test-Programm unterteilt durch eine Menü-Führung die einzelnen zu testenden Abschnitte.

2.1. Firmware Test Spezification

- Es soll die Programm-Übertragunge getestet werden. Dafür soll sowohl ein beliebiges C++, als auch ein Python Programm übertragen werden.
- Es sollen alle Python-Funktionen getestet werden. Der Fokus ist hier auf korrekter Parameter-Übergabe. Die Rückgabe-Werte von Funktionen werden auf dem Monitor ausgegeben. Hier sollen im Wesentlichen verschiedene Devices in den verschiedenen möglichen Konfigurationen getestet werden.
- Es sollen die Exit-Status-Ausgaben und das korrekte Handhaben von Exceptions überprüft werden. Dazu gehört das Ausgeben von Fehlermeldungen und die Wieder-Ausführbarkeit der MP-VM nach einer Exception. Es soll auch der Hard-Fault-Handler getestet werden.
- Es sollen die zusätzlich eingebundenen Buildin-Funktionen überprüft werden. Für eingebundene Module reicht hier die Verfügbarkeit, solange diese nicht im Rahmen dieses Projektes entstanden sind, so z.B. das Math-Modul.

2.2. Firmware Tests

Die Firmware-Tests sind als interaktive Tests gedacht. Es können alle Funktionen des ORB aufgerufen und ausgegeben werden. Schon durch die Möglichkeit das Test-Programm zu erstellen, ist die Funktions-Fähigkeit der ORB-Firmawre zu einem großen Teil vallidiert. Das restliche Testen, Neben der erreichbarkeit der Funktionen, ist das korrekte Übergeben von Parametern. Um dies testen zu können ,soll das Firmware-Test-Programm verwendert werden. Es bietet die Möglichkeit an, die Python-Funktionen zu mit verschiedenen Parametern aufzurufen. Hier wurden alle Parameter-Variationen für die möglichen Funktions-Aufrufe getestet. Im Folgenden sind zusätzliche Informationen für den Vorgang der Tests. Alle im folgenden aufgezählten Tests konnten erfolgreich durchgeführt werden.

2.2.1. Program Upload

Für den Program-Upload-Test wurde das C++-Demo-Programm übertragen. Die Ausführung von diesem ist identisch zu dem C++-Programm mit alter Firmware. Die C++-Programmübertragung konnte vallidiert werden. Das Aufspielen des Python-Firmware-Test-Programms war auch erfolgreich. Es konnte also sichergestellt werden, dass die ORB-Firmware sowohl Python-Programme als auch C++-Programme erfolgreich übertragen kann. Außerdem ist durch diesen Test überprüft worden, dass die ORB-Firmware zwischen Python-Programm und C++-Programm unterscheiden kann. Das Python-Programm und das C++-Programm lassen sich sowohl über die ORB-Monitor-Tasten, sowie die ORB-On-Board-Tasten starten und stoppen. Die Status-LEDs des ORB werden dabei korrekt gesetzt.

2.2.2. Modules

2.2.2.1. Devices - Servo Module

Der Servo-Module-Test erlaubt es einem Nutzer alle Parameter eines Servo-Objektes einzustellen. Es sind der Port, die Geschwindigkeit und der Winkel einstellbar. Hier kann auch optional mit einer LED getestet werden, da ein Sero-Motor mit Hilfe eines PWM-Signales angesteuert wird. Es konnte validiert werden, dass Servo-Motoren konfigurierbar sind.

2.2.2.2. Devices - Motor Module

Der Motor-Test ist in drei Teile unterteilt: 'config', 'set', 'get'. Dies sind die drei Funktionen, welche durch ein Motor-Objekt verwendbar werden. Der erstellte Motor-Test erlaubt es einen Nutzer einen Lego-Motor vollständig zu konfigurieren. Einstellungen wie 'kp', 'ki', etc., also motor-spezifische Konfigurationen sind hier für den LEGO-Motor eingestellt. Diese wurden in dem Line-Follower mit anderer konfiguration verwendet. Somit ist die Konfiguration für einen Lego-Motor und einen MakeBlock-Motor vorgenommen und überprüft worden. Alle Funktionen ließen sich vallidieren.

2.2.2.3. Devices - Sensor Module

Der Sensor-Test ist in dre Teile aufgeteilt: 'config', 'get', 'pre-configured'. 'get' und 'config' werden verwendet um Sensoren zu benutzen. Hier werden die Sensor-Objekt-Funktionen abgebildet. Über 'pre-configured' wird ein voreingestellter Sensor verfügbar. Hier wurde ein NXT-Light-Sensor vor-konfiguriert und durchläuft Test-Funktions-Aufrufe und Test-Konfigurationen. In diesem Test sollen verschiedene Sensoren an dem ORB angeschlossen und konfiguriert werden. Es wurden Tests für MB-Color-Sensor, MB-Ultra-Sonic-Sensor, NXT-Light-Sensor und NXT-Touch-Sensor durchgeführt. Im Wesentlichen wurde hier alle verschiedenen Funktions-Parametern für Sensoren verwendet. Dadurch wurde vallidiert das diese korrekt konfigurierbar sind.

2.2.2.4. Memory

Der alternative Programmstart mit "Write Mem" kann verwendet werden um einen Wert in den Nutzer-Speicher zu schreiben. Mit dem Programm-Start "Tests" kann unter 1.Module->2.Memory die GetMemory-Funktion und ClearMemory-Funktion getestet werden. GetMemory gibt dabei den durch "Write Mem" in den Nutzer-Flash-Speicher geschriebenen Wert aus. Clear-Memory setzt den vollständigen Nutzer-Speicher zu 'xFF' zurück, darin inbegriffen den vorher geschriebenen Wert. Die Funktionalität der Memory-Funktionen konnte durch diesen Test sicher gesetellt werden.

2.2.2.5. Time

Test-Name	Test-Vorgang	Verhalten
Get Time	Ausführen.	Es wird der Wert von getTime wiedergegeben.
wait	Ausführen.	Der Nutzer wird darüber Informiert, dass 2 Sekunden gewartet wird. Danach wird "Finished" ausgegeben.

Abbildung 12: Time

2.2.3. Builtin Funktions

Test-Name	Test-Vorgang	Verhalten	Notiz
Math	-	Die erste Zeile dieses Tests soll ausgeben, ob das Math-Module verfügbar ist.	
Exit	Ausführen.	Die MP-VM stoppt ihr Ausführung. Auf dem ORB-Monitor wird ein System-Exit angezeigt.	
Min/Max	Ausführen. Mit Buttons Werte auswählen.	Auf dem Monitor wird das Minimum und Maximum der beiden Werte angezeigt.	
getArg	Ausführen.	Der Arg-Wert wird ausgegeben. Dieser ist 0.	

Abbildung 13: Builtin Funktions

2.2.4. Error-Tests

Test-Name	Test-Vorgang	Verhalten	Notiz
Hard-Fault	Starte einen Motor mit dem Motor-Test. Geh zurück zum "Main Menu". Danach Hard-Fault-Test ausführen.	ORB-Firmware unterbricht Ausführung. Alle 3 Status LEDs gehen an. ORB-Firmware muss resettet werden. Der zuvor gestartete Motor wird gestoppt.	Der Hard-Fault ist schwer zu produzieren, nach Änderungen an der Firmware könnte dieser Test nicht mehr funktionieren. Die aktuelle Firmware-Version produziert diesen jedoch korrekt. Es wurde mit der Hardware-Version 0.22 getestet.
Memory-Error	Ausführen	Die MP-VM unterbricht die Ausführung. Es wird eine Exception-Message auf dem ORB-Monitor ausgegeben. Diese weißt auf einen Memory-Error hin.	
User-Exception	Ausführen	Die MP-VM unterbricht die Ausführung. Es wird eine Exception-Message auf dem ORB-Monitor ausgegeben. Diese weißt	

Test-Name	Test-Vorgang	Verhalten	Notiz
		auf eine User-Exception hin.	
User-Interrupt	Ausführen, danach muss der Stop-Button des ORB gedrückt werden.	Die MP-VM unterbricht die Ausführung. Es wird der 'User Interrupt' auf dem ORB-Monitor ausgegeben.	

Abbildung 14: Error-Tests

Evaluation

Inhaltsverzeichnis

Evaluation

Die in den Anforderungen definierten Ziele konnten vollständig umgesetzt werden. Durch die Windows- und Firmware-Tests, sowie das Erstellen der Benchmark-Programme, konnte die grundsätzliche Funktionsfähigkeit der MP-VM und Anbindung bestätigt werden. Auch alle in den Reviews besprochenen bzw. gewünschten Änderungen an der ORB-Firmware, konnten umgesetzt werden. Daher ist im Hinblick auf diese Punkte, diese Arbeit seitens technischer Anforderungen ein voller Erfolg. Das Benchmarking konnte auch die MP-VM-Performance der C++-Performance gegenüberstellen. Es gibt einige Anwendungen, die durch die Verwendung der MP-VM deutlich langsamer werden oder unter Umständen gar nicht korrekt funktionieren könnten. Dies war jedoch zu erwarten. Die Verwendung eines Interpreters wird in jedem Fall langsamer als eine C++-Implementation laufen. Es konnte jedoch auch gezeigt werden, dass es Anwendungen gibt, welche durch das Verwenden der MP-VM nicht deutlich eingeschränkt werden. So wird es viele Anwendungen geben, welche problemlos ihre Funktionen erfüllen können. Die verschiedenen Benchmark-Programme wurden miteinander verglichen und als Gesamtheit betrachtet. Abschließend wurden die Ergebnisse zusammengeführt und eine entsprechende Schlussfolgerung gezogen. Diese findet sich im Implementierungsreport unter 1.6.5. Schlussfolgerung. Im Laufe der Implementierung konnten viele Probleme identifiziert und gelöst werden. Hier ist anzumerken, dass es ein paar Probleme gibt, deren Lösung den Umfang dieser Bachelorarbeit überschritten hätten, wie z.B. das Einbinden von Middleware über ein File-System. Für diese auftretenden Probleme wurde jedoch in jedem Fall eine Lösung gefunden, welche eine stabile Verwendung der ORB-Firmware gewähreisten kann. Es wird auch mit solchen Problemen umgegangen, die nicht unbedingt lösbar sind, wie der Umgang mit Hard-Faults, welche durch die MP-VM produziert werden können. Probleme, welche nicht zur Gänze gelöst werden konnten, werden im Ausblick weiter ausgeführt. Ebenso weitere Verbesserungsvorschläge, die diese Arbeit noch verfeinern könnten oder auch in einer Anschluss-Arbeit umgesetzt werden könnten. Zusammengefasst wurden alle Anforderungen erfüllt. Es liegt eine stabile ORB-Firmware mit den gewünschten Funktionen und API-Dokumentationen vor. Die ORB-Application wird durch die MicroPython-VM vollständig abgebildet. Die Programmierung mit MicroPython ist konzeptionell mit der Programmierung in C++ gleichzustellen. Es wird Middleware und das Verwenden des ORB-Monitors unterstützt.

Ausblick

Im Ausblick werden weitere angedachte Verbesserungsmöglichkeiten vorgestellt.

Performance Verbesserung

Die MP-VM hat im Vergleich zu der C++-Application je nach Anwendung eine vergleichbar schlechte Performance. Gerade solche Funktionen welche die Get-Funtion für Sensor- und Motor-Daten verwenden. Hier wird im Moment ein Dictionary aufgebaut. Dies ist gewählt worden, da es sich um einen bereits in MicroPython integrierten Typ handelt. Hier kann jedoch vermutet werden, das ein Dictionary zu mächtig für diesen Anwendungsfall ist. So könnte man versuchen Sensor- und Motor-Report-MicroPython-Objekte anzulegen. Diese würden dann für genau diesen Zweck umgesetzt sein. Diese könnten den Overhead, der mit dem komplexen Dictionary-Objekt zusammenhängt, minimieren. Hier handelt es sich jedoch eher um eine Vermutung, als eine Anleitung die Performance zu verbessern. Die Annahme stützt sich darauf, dass ein spezialisiertes Objekt besser optimiert werden kann. Es können so z.B. Typ-Checks umgangen werden, wenn der Nutzer nicht in der Lage ist dieses Objekt zu erstellen. Das Sensor- oder Motor-Report-Objekt würde also nur durch die Rückgabe einer Funktion instanziiert werden können. Eine weitere Übelegung, wäre das Verwenden einer anderen MicroPython-Objekt repräsentation. Hier müssten die Vor- und Nachteile der MicroPython-Objekt-Repräsentationen gründlich überlegt werden. Wie bereits, in Konzepte: Wie funktioniert die Typ-Zuordnung (Konkrete-Typen), erwähnt wird im Moment 'MICROPY_OBJ_REPR_A' verwendet. Die anderen MicroPython-Objekt-Repräsentation Funktionieren konzeptionell gleich. Die Modul-Implementationen müssten nicht angepasst werden. Je nach gewählter Repräsentation könnte ein größerer Heap-Speicherverbrauch zugunsten einer besseren Performance in Kauf genommen werden.

File-System in die ORB-Firmware Integrieren

Das Integrieren eines File-Systems in die ORB-Firmware wäre eine Verbesserung die man vornehmen könnte. Diese bringt, zum aktuellen Stand Probleme mit sich. Der Flash-Speicher des ORB ist sehr limitiert. Daher wurde zu diesem Zeitpunkt davon abgesehen. Jedoch würde die Integration Vorteile mit sich bringen. Python-Programme könnten auf dem File-System basierend eine Import-Struktur verwenden. Dies würde Nutzern erlauben MicroPython für das ORB zu schreiben, ohne die '.build'-Datei zu benötigen. Import-Abhängigkeiten wären so klarer strukturierbar. Um dies umzusetzen, muss klar bestimmt werden wie viel Overhead durch ein File-System entsteht. In diesem Schritt wäre es möglich zu überlegen, das ORB um einen externen Speicher zu erweitern. Hier müsste zu allererst überlegt werden, welches Speicher-Medium gewählt werden soll. So könnte ein externer Flash-Speicher fest auf das Board integriert werden. Falls diese Änderung in Betracht gezogen wird, so muss sichergestellt werden, dass auch das C++-Programm von dem Speicher-Medium aus ausgeführt werden kann.

Peripherie durch Middleware abbilden

Es gibt eine Vielzahl an Peripherie, welche durch das ORB unterstützt wird. In diesem Projekt wird durch 'Build'-Dateien eine Möglichkeit geboten, mehrere Python-Scripts zu einem Programm zu kompilieren. Dieser Prozess kann genutzt werden um sogenannte Middleware umzusetzen. Dies ist ein durch die ORB-C++-Programme eingeführtes Konzept. Es sollen zusätzliche Klassen angeboten werden, welche verschiedene Peripherien für eine nutzerfreundliche Verwendung bereitstellen. So sollten auch für die Python-Umgebung zusätzliche Klassen und Module umgesetzt und als eine Middleware angeboten werden.

Code-Editor in ORB-Monitor integrieren

Durch die Erweiterung der ORB-Firmware um die MP-VM wurde ein Build-Prozess für die Python-Programme mithilfe des ORB-Util umgesetzt. Dies ist jedoch ein für dieses Projekt spezifischer Prozess. Ein Nutzer muss sich mit dem Command-Line-Tool vertraut machen um Python-Programme zu programmieren. Das ORB-Util ist zwar ein nutzerfreudnliches Tool, aber man könnte diesen Vorgang noch verbessern. Eine Möglichkeit wäre das Integrieren eines Code-Editors in den ORB-Monitor. Dieser könnte einfach das ORB-Util zum Kompilieren von Python-Programmen nutzen. Die Projekt-Struktur, also im Prinzip die `.build`-Datei, könnte dabei durch den ORB-Monitor abgebildet und generiert werden. Der Nutzer müsste hier im besten Fall die Möglichkeit haben, Python-Dateien anzulegen und diese in Ordnern ablegen zu können. Im besten Fall hätte die Editor-Oberfläche eine Python-Syntax-Highlighting-Unterstützung. Durch die Integration eines Code-Editors in die ORB-Firmware wäre es für einen Nutzer möglich, Python-Programme für das ORB zu schreiben, ohne ein zustätzliches Tool zu lernen. Hier könnte man sich von MicroPython-Editoren wie zum Beispiel 'Thony' inspirieren lassen. Dadurch das die MP-VM auch unter Windows getestet werden kann, könnte man hier auch das Debuggen unter Windows anbieten.

Ausgabe von Print Anweisungen und Fehler-Meldungen

Die Ausgaben von Print-Statements und Error-Stacktraces stellten sich als eine Herausforderung dar. Die Schwierigkeit Micropython-Errors auszugeben, liegt an durch den ORB-Monitor vorgegebenen Limitierungen. Hier ist sowohl die Länge als auch die Formatierung der Ausgabe ein Problem. Wie bereits im Implementierungsreport beschrieben, ist die MP-VM-Print-Funktion darauf ausgelegt Strings so auszugeben, wie man es über UART erwarten könnte. So werden u.A. die Ausgabe von Arrays also z.B. `"[1,2,3,4,5]"` dadurch realisiert, dass das Array als Einzelteile ausgegeben wird. So bekommt man von der Micropython Print-Funktion, in diesem Fall `"[" "1, " "2, " "3, " "4, " "5, " "]\r\n"` also alle Einzelteile der Ausgabe, als einen eigenen String übergeben. Dieses Problem würde sich durch eine UART-ähnliche Ausgabe über den ORB-Monitor lösen. Also durch eine wachsende oder scrollbare Text-Box und nicht etwas die 4 vorgegebenen Zeilen. Dadurch wäre es auch möglich größere Ausgaben wie die von Fehler-Meldungen mit den zusätzlichen Informationen über den Stacktrace zu ermöglichen. Der ORB-Monitor würde bei dieser Lösung die Print-Ausgabe wie durch MicroPython vorgegeben bekommen. Alternativ könnte man die Micropython-VM Fehler-Meldung im Flash-Speicher ablegen und von dem ORB-Monitor als eine Text-Datei auslesbar machen. Die Anpassung des ORB-Monitor-Text-Feldes als Lösung erscheint jedoch am nutzerfreundlichsten.

MP-VM kann Hard-Faults produzieren

Durch die MP-VM ist in die ORB-Firmware eine Komponente integriert worden, welche Speicher-Zugriffs-Fehler produzieren kann. Die MP-VM macht in einigen Fällen nicht ausreichende Überprüfungen ob, zum Beispiel, noch genügend MP-VM-Heap-Speicher zur Verfügung steht. Um mit diesem Problem umzugehen wurde ein Fault-Handler in die ORB-Firmware aufgenommen. Dies ist jedoch ein Problem, welches vermutlich nicht zu lösen ist. Denn für diesen Zweck müsste das MicroPython-Projekt vollständig überarbeitet werden. Es müssten an allen Stellen, in denen ein Objekt in dem Heap-Speicher angelegt oder verändert wird, zusätzliche Überprüfungen eingebaut werden. Dies wäre ein massiver Aufwand und würde die Performance der MP-VM stark negativ beeinflussen. Falls der Nutzer auf diese Probleme trifft, kann er jedoch selbst Garbage-Collect-Zyklen in sein Projekt einbauen. Der beste Umgang mit diesem Problem ist wahrscheinlich ein regelmäßiges Re-Basen des MicroPython-Submodules auf dem aktuellen Release-Branch des MicroPython-Projektes. So könnte das ORB-Projekt durch neue Bugfixes und Performance-Verbesserungen des MicroPython-Projektes Profitieren.

Integration von Threading

Die Integration von Funktionen, welche das Verwenden von Threads erlauben, wären von Vorteil. Hier könnte man sich an bereits bestehenden MicroPython-Ports orientieren. Es gibt MicroPython-Ports wie zum Beispiel den Raspberry-Pi-Pico-Port. Diese Ports bieten eine einfache Variante von Threads an. So kann der Nutzer nicht selber Threads erstellen. Es wird eine statische Anzahl an Threads unterstützt. Der Nutzer kann den Threads nur Funktionen zuweisen, diese starten und stoppen. Die Integration solcher einfachen Threads in die ORB-C++-Application sollte dabei relativ gradlinig sein. Hier könnte man pro angebotenen Thread eine Task erstellen. Hier würde dem Nutzer die Möglichkeit gegeben werden, die Update-Funktion eines Threads zuzuweisen. Die `orblocal.h` würde damit zum Beispiel um folgendes erweitert werden:

```
void threadStart(int id);
void threadStop(int id);
```

```
void threadSetFunction(int id, void* fun);
```

Es muss jedoch auch beachtet werden, dass in den ORB-Funktionen Race-Conditions berücksichtigt und entsprechend behandelt werden. Wie zum Beispiel in der Set-Motor-Funktion. Ohne weitere Anpassungen würde hier eine Race-Condition auftreten, da auf eine gemeinsame Motor-Task geschrieben wird. Je nach Reihenfolge der Operation könnte so ein ungewollter Motor-Zustand entstehen. Die Integration von Threading wird von der MicroPython-VM unterstützt. Für diesen Zweck müssen für das Threading benötigte Funktionen umgesetzt werden. Diese werden von der MicroPython-VM vorgegeben. Ein guter Einstiegspunkt wäre hier das Umsetzen einer 'mpthreadport.h'-Datei. Hier wird eine Vielzahl an für das Threading benötigten Funktionen der MicroPython-VM erwartet. Hier sind vermutlich noch weitere Änderungen notwendig. So muss auch die ORB-Firmware und MicroPython-Konfiguration angepasst werden.

> Für den Raspberry-Pi-Pico-Port findet sich diese Datei unter: micropython/ports/rp2/mpthreadport.h.

Da alle Threads für die MicroPython-VM eine geteilte VM haben, müsste die ORB-Firmware hier um Konzepte, wie zum Beispiel Mutex, erweitert werden. Dies ist jedoch mit erheblichem Implementierungsaufwand verbunden. Es kann sich an der Vielzahl an bestehenden MicroPython-Ports orientiert werden.

Tools und Referenzliste

Tools und Referenzen

Tool	Version	Kommentar
Git	2.38.1.windows.1	
Code::Blocks	20.03	
EmBitz	2.50	
Python	Python 3.11.4	
MSYS2	20240507	MSYS2 will be used to install gcc and all the needed tools, such as `make` and `GNU coreutils`.
GCC	13.2.0 (Rev3, Built by MSYS2 project)	installed by MSYS2
Make	GNU Make 3.81	installed by MSYS2
DFU-Util	0.7	
zadig	2.9	
Sphinx	8.0.2	
chocolatey	2.2.0	
ORB-Firmware	https://github.com/ThBreuer/ORB-Firmware Commit: e8d74fb	This Repository was integrated into this Project, and added as a Submodule.
ORB-Application	https://github.com/ThBreuer/ORB-Application Commit: f0a4fbb	This Repository was integrated into this Project, and added as a Submodule.
MicroPython	https://github.com/micropython/ Commit: e9814e9	This Repository was integrated into this Project, and added as a Submodule.
Mpy-Cross	v6.3	Part of the MicroPython Repository
Python-Intelhex	https://github.com/python-intelhex/intelhex Commit: 6d0e826	
bin2hex.py	https://gist.github.com/pavel-a/89d71b3aba9d7a9e6f8a61d728b08a8e	

Literatur und Referenzen

Kürzel	Autor	Quelle
ORB-FW	Thomas Breuer. :	„ORB-Firmware“. https://github.com/ThBreuer/ORB-Firmware Commit: e8d74fb
ORB-APP	Thomas Breuer. :	„ORB-Application“. https://github.com/ThBreuer/ORB-Application

Kürzel	Autor	Quelle
		Commit: f0a4fbb
MP	Damien P. George, Paul Sokolovsky et al. :	„MicroPython“. https://github.com/micropython. Commit: e9814e9. Datum: 16.08.2024.
MPD	Damien P. George, Paul Sokolovsky et al. :	„Implementing a Module“. https://docs.micropython.org/en/latest/develop/library.html Stand: 23.10.2024.
MPC	Peter Hinch. :	„Exit micropython from interrupt in c“. https://forum.micropython.org/viewtopic.php?t=2521#p14831 Datum: 17.10.2016.
ARM-al	Hrsg.: Arm Limited :	„Basic data types“. https://developer.arm.com/documentation/dui0491/i/C-and-C---Implementation-Details/Basic-data-types Stand: 30.10.2024
C-s	Hrsg.: ©ISO/IEC, Ballot-Version, Zugriff über: The University of Western Australia.:	„Programming languages — C“. ISO/IEC 9899:2017 - Ballot C17 https://teaching.csse.uwa.edu.au/units/CITS2002/resources/n2176.pdf Jahr: 2017.
-	Hrsg.: Arm Limited.:	„Arm® Cortex®-M4 Processor Technical Reference Manual“. https://documentation-service.arm.com/static/5f19da2a20b7cf4bc524d99a Stand: 02.Mai.2010
-	Joseph Yiu. :	„The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors“. ISBN: 9780124079182 Datum: 06.10.2013
-	Jonathan W. Valvano.:	„Embedded Systems: Real-Time Operating Systems for Arm Cortex M Microcontrollers“. ISBN: 978-1466468863 Datum: Januar 2017
GCC-cf	Hrsg.: Free Software Foundation, Inc.:	„3.11 Options That Control Optimization“. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html Stand: 31.10.2024
GCC-co	Hrsg.: Free Software Foundation, Inc.:	„6.3.3.2 Compilation options“. https://gcc.gnu.org/onlinedocs/gnat_ugn/Compilation-options.html Stand: 31.10.2024
-	Hrsg.: STMicroelectronics.:	„STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs“. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf Datum: Juni 2024

Kürzel	Autor	Quelle
STM	Hrsg.: STMicroelectronics.:	„STM32 Cortex®-M4 MCUs and MPUs programming manual“. https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf Datum: März 2024
AMD	Hrsg. Advanced Micro Devices, Inc.	„AMD64 Architecture Programmer’s Manual Volume 1: Application Programming“. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24592.pdf Datum: Oktober 2020

ORB-Python Documentation

Nils Hoffmann

Nov 08, 2024

CONTENTS

Welcome to the documentation of the **ORB-Python** project.

CHAPTER

ONE

PYTHON API

This part of the Documentation is the Python API. Here you can find how to program your Open Robotics Board using Python.

1.1 devices

The Devices Module allows you to program the ORBs configurable periferals as well as onboard devices.

1.1.1 sensor

The *Sensor* class represents a sensor device with configurable parameters and multiple data retrieval methods.

class devices.**sensor**(*port: int*, *type: int*, *mode: int*, *option: int*)

Initializes a sensor instance connected to a specified port with defined type, mode, and optional parameters.

Parameters

- **port** (*int*) – The port number the sensor is connected to.
- **type** (*int*) – Type of the sensor, such as *Analog*, *I2C*, *TOF*, *Touch*, or *UART*.
- **mode** (*int*) – Operating mode of the sensor.
- **option** (*int*) – Additional configuration option.

config(*type: int*, *mode: int*, *option: int*)

Configures the sensor.

Parameters

- **type** (*int*) – Type of the sensor (*Analog*, *I2C*, *TOF*, *Touch*, or *UART*).
- **mode** (*int*) – Operating mode of the sensor.
- **option** (*int*) – Additional configuration option.

get() → Dict["values": [int ,int], "type": int, "option": int, "lenExp": int]

Retrieves the sensor report as a dictionary representation.

getDigital() → int

Returns a digital reading from the sensor.

Returns

The digital value.

Return type
int

`getValueExt`(*channel: int*) → int

Parameters
channel (*int*) – The channel number to retrieve the value from.

Returns
The value from the specified channel.

Return type
int

Ports

`S1 = 0`
Sensor port 1.

`S2 = 1`
Sensor port 2.

`S3 = 2`
Sensor port 3.

`S4 = 3`
Sensor port 4.

Types

`Analog = 0`
Analog sensor type.

`I2C = 1`
I2C sensor type.

`TOF = 2`
Time-of-Flight sensor type.

`Touch = 3`
Touch sensor type.

`UART = 4`
UART sensor type.

1.1.2 motor

The *Motor* class represents a motor device with configurable parameters and various modes of operation.

`class devices.motor`(*port: int*, *direction: int*, *ticks: int*, *acc: int*, *kp: int*, *ki: int*)
Initializes a motor instance.

Parameters

- **port** (*int*) – The port that the motor is connected to.
- **direction** (*int*) – Direction of the motor (*FORWARD* or *REVERSE*.)
- **ticks** (*int*) – Number of encoder ticks for one revolution.
- **acc** (*int*) – Acceleration rate of the motor.

- **kp** (*int*) – Proportional gain for the motor control.
- **ki** (*int*) – Integral gain for the motor control.

config(*direction: int*, *ticks: int*, *acc: int*, *kp: int*, *ki: int*)

Configures the motor parameters.

Parameters

- **direction** (*int*) – Direction of the motor, either *FORWARD* or *REVERSE*.
- **ticks** (*int*) – Number of encoder ticks for one revolution.
- **acc** (*int*) – Acceleration rate.
- **kp** (*int*) – Proportional gain.
- **ki** (*int*) – Integral gain.

set(*mode: int*, *speed: int*, *position: int*)

Sets the motor's operating mode, speed, and position.

Parameters

- **mode** (*int*) – The mode in which to operate the motor (*POWER_MODE*, *BRAKE_MODE*, *SPEED_MODE*, or *MOVETO_MODE*).
- **speed** (*int*) – The speed of the motor.
- **position** (*int*) – The target position for the motor.

get() → Dict["speed": int, "power": int, "position": int]

Returns

A dictionary containing: - **speed**: The current speed. - **power**: The current power. - **position**: The current position.

Ports

M1 = 0

Motor port 1.

M2 = 1

Motor port 2.

M3 = 2

Motor port 3.

M4 = 3

Motor port 4.

Modes

FORWARD = 1

REVERSE = -1

POWER_MODE = 0

BRAKE_MODE = 1

SPEED_MODE = 2

MOVETO_MODE = 3

1.1.3 servo

The *Servo* class represents a servo device with specific parameters.

class devices.**servo**(*port: int*)

A servo object.

Parameters
port (*int*) – The port number the servo is connected to.

set(*speed: int*, *angle: int*)

Sets the speed and angle for the servo.

Parameters

- **speed** (*int*) – Speed of the servo movement.
- **angle** (*int*) – Target angle for the servo.

Ports

S1

Servo port 1.

S2

Servo port 2.

1.2 memory

The *memory* module provides functions to manage memory operations.

memory.**setMemory**(*addr: int*, *data: list | bytes*)

Sets memory at the specified address with the given data.

Parameters

- **addr** (*int*) – The memory address to set.
- **data** (*list | bytes*) – The data to be stored, which can be a list or bytes.

memory.**getMemory**(*addr: int*, *length: int*) → bytes

Returns memory starting from the specified address in the user-memory-region.

Parameters

- **addr** (*int*) – The starting memory address.
- **length** (*int*) – The number of bytes to retrieve.

Returns
The retrieved memory content as bytes.

Return type
bytes

memory.**clearMemory**()

Clears all stored memory data.

1.3 time

The *time* module provides utilities for working with time, including retrieving the current time and introducing delays.

`time.getTime() → int`

Returns the time, since microcontroller start.

Returns
The current time in milliseconds.

Return type
int

`time.wait(ms: int)`

Pauses execution for a given number of milliseconds.

Parameters
ms – Number of milliseconds to wait.

Type
int

1.4 monitor

The *monitor* module provides functionality for ORB-Monitor communication.

`monitor.getKey() → int`

Returns
Current key pressed on the monitor.

Return type
int

`monitor.setText(text: str)`

Sets the text to be displayed on the monitor. At the Moment only 32 characters are supported for one Line.

Parameters
text (*str*) – The text to display.

`class monitor.keys`

A class representing key constants for the *monitor*.

Each key constant has a unique integer value.

Available keys:

- **NO_KEY** = 0
- **A1** = 1
- **A2** = 2
- **A3** = 3
- **A4** = 4
- **A5** = 5
- **A6** = 6

- **A7** = 7
- **A8** = 8
- **B1** = 9
- **B2** = 10
- **B3** = 11
- **B4** = 12
- **B5** = 13
- **B6** = 14
- **B7** = 15
- **B8** = 16
- **B9** = 17
- **B10** = 18
- **B11** = 19
- **B12** = 20
- **C1** = 21

CHAPTER

TWO

VM API

class **PythonVM**

A class to manage the Micropython-VirtualMachine. Providing methods for running, stopping, and retrieving the VM's status.

void PythonVM::**run**(LoadLengthFunction loadLength, LoadProgramFunction loadProgram, uint8_t arg)

Starts the VM with the specified load functions and argument.

Parameters

- **loadLength** – Function pointer that returns the program length
- **loadProgram** – Function pointer that loads the program data
- **arg** – Additional argument passed to the program

bool PythonVM::**isRunning**()

Checks if the VM is currently running.

Returns
True if the VM is running, otherwise False

Rtype
bool

void PythonVM::**stopProgram**()

Stops the currently running program in the VM.

int PythonVM::**getExitStatus**()

Retrieves the exit status of the VM.

Returns
Exit status code

Rtype
int

const char *PythonVM::**getExitInfo**()

Returns additional information about the VM's exit status.

Returns
Exit information string

Rtype
const char*

enum **Status**

Enumeration for VM exit statuses, indicating various exit conditions.

Status codes:

- **NORMAL**: Program exited normally.
- **EXCEPTION**: Program exited with an exception.
- **INTERRUPT**: Program was interrupted by User.

INDEX

README

This is a simple documentation to set up the project and provide a concise explanation of what this project is about. The in-depth documentation is only available in German and can be found here.

The goal of this project is to integrate a MicroPython interpreter into the firmware of the Open Robotic Board (ORB). The ORB is a hardware platform that enables the modular construction of a robot. The ORB-Firmware provides functions, such as motor control, for robot programming. The aim is to make it possible to develop Python programs for the ORB, with a connection to the ORB-Monitor program. The ORB-Monitor program is a development tool that allows text output, as well as the uploading, starting, and stopping of programs. The existing functionalities of the ORB, such as the development of C++ programs, should remain functionally the same.

> Note
>
> The ORB uses the STM32F405 microcontroller, and schematics for the board are open to the public. All information about the board, additional software, supported sensors, and the original firmware can be found at: https://github.com/ThBreuer/OpenRoboticBoard. This project is based on the design and work of Thomas Breuer regarding the ORB-Projects.

The following steps will guide you through the project setup. If you only want to develop and run Python projects on your ORB, skip to User Environment.

Project Setup

- Setup Code::Blocks Windows Debug Environment
- Building with Code::Blocks
- Setup EMbitz Firmware Development Environment
- Flash the Firmware
- Develop a Python Program
- Develop a C++ Program

Setup Code::Blocks Windows Debug Environment

1. (Install Git and) Clone the project:

   ```
   git clone https://github.com/NiHoffmann/ORB
   ```

2. Download and install Python 3. Make sure to add Python 3 to your environment variables, so the python3 command is accessible. To check if correctly installed, execute:

   ```
   python3 --version
   ```

3. Install the GCC compiler, make, and rm (utils). I suggest using Msys2 for this installation. It will come prepackaged with all the needed utilities for this project. Make sure to add the Msys2 bin folder to your PATH. To check if correctly installed, execute:

   ```
   gcc --version
   ```

> Note

> If you already have Cygwin installed, it might interfere with the Msys2 installation process. If you encounter any problems, try to remove Cygwin from your Windows PATH variable. You may add it back after the installation is complete.

4. Initialize the MicroPython submodule:

```
git submodule init
git submodule update
```

5. Build the mpy-cross compiler:

```
cd <ORB-PATH>/micropython/mpy-cross
make
```

   or install it via pip:

```
pip install mpy-cross
```

   Make sure to add mpy-cross to your PATH. To check if correctly installed, execute:

```
mpy-cross --version
```

6. Open the Code::Blocks project.

   i. Select the GNU GCC compiler and make sure you have the `-Og` compiler flag set. (This will suppress potentially application-breaking aggressive optimization.)
   ii. Select the "Rebuild" target. This will build the project and rebuild the MicroPython embedded port.

7. Add `<ORB-PATH>/ORB-Python/tools` to your Windows PATH variable.

Building with Code::Blocks

There are two build targets for this project: Build & Rebuild. Both targets, when built, compile the file `program.py` located in `<ORB-PATH>/ORB-Python/Program`. This file will be executed by the MicroPython interpreter; you can write your Python code for testing here.

The differences are as follows:

- **Build**: Only compiles the Code::Blocks project part. This can be used if there were no QStrings changed (e.g., adding or changing names for functions, modules, or constants). Use this build target to save on compile time.
- **Rebuild**: Also regenerates the MicroPython embedded library, registering changes made to the MicroPython interpreter.

> Important
>
> Just running the project will not recompile the `program.py` file. It is suggested to always use the "Build and Run" function.

Setup EMbitz Firmware Development Environment

Building the firmware is currently only supported in a Windows environment.

1. Follow the steps described in Setup Code::Blocks Windows Debug Environment for a basic project setup.
2. Install EMbitz.
3. Execute the `_setEnv.bat` inside the ORB-Firmware folder.
   > This will set the environment needed for working with the EMB-SysLib.
4. Go to `<ORB-PATH>/ORB-Firmware/project` and start the EMbitz project.
5. In the build options of this project, make sure "GNU Arm Embedded Toolchain" is selected as the compiler.
   > Commonly known as "gcc-arm-none-eabi"; this should come prepackaged with the EMbitz IDE.
6. Select the "Build All" target and rebuild the whole project.

Flash the Firmware

Build the firmware using the EMbitz project. Alternatively, you can download it from the releases. The release will also include tools to flash the firmware. You may ignore the steps after this and follow the instructions described in the release README.

You may flash the firmware with a tool of your choice. There is a copy of the DfuSe tool at `ORB/ORB-Firmware/Tools/DfuSe/Bin`. Additionally, `dfu-util` is available at `ORB/ORB-Python/tools`. Alternatively, you can use the `orb-util` for this. If you want to use `orb-util`, make sure to change the DFU device driver to WinUSB. An easy-to-use tool for this is "Zadig," also available in the tools folder. Put your ORB into DFU-Mode and Execute:

```
orb-util --flash <HW-Version>
```

To flash your Firmware.

> **Note**
>
> To put the ORB in DFU mode, plug a jumper pin into the pins labeled BOOT0 (the outer 2 pins of the 3 present).

Develop a Python Program

There are two ways to compile and run a Python program.

1. Development Environment

If you followed the steps to set up the ORB project, you are ready to go. The `orb-util` tool will be available as a command. You can use this to set up and compile MicroPython-Projects. Alternatively, you can use the Windows environment for debugging, as described in Building with Code::Blocks.

2. User Environment

i. Download the project release.
ii. Follow the instructions described in the release README.

This will allow you to develop Python programs without needing to set up the ORB-Project.

Develop a C++ Program

Go to `ORB/ORB-Application` and execute `_setEnv.bat`. Now you can check out the example projects. Use the different `.bat` files to access documentation, the ORB-Monitor (upload tool), clean the project, or start EMbitz for developing your project. Just like developing a Python program, these are starting points to figure out the workflow.

> **Note**
>
> This is not part of my work, but I still feel it's worth mentioning. Although I did some modifications to the ORB-Application (e.g., adding language flags to the linker script and moving the program ROM address to the same location as the Python program).