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.

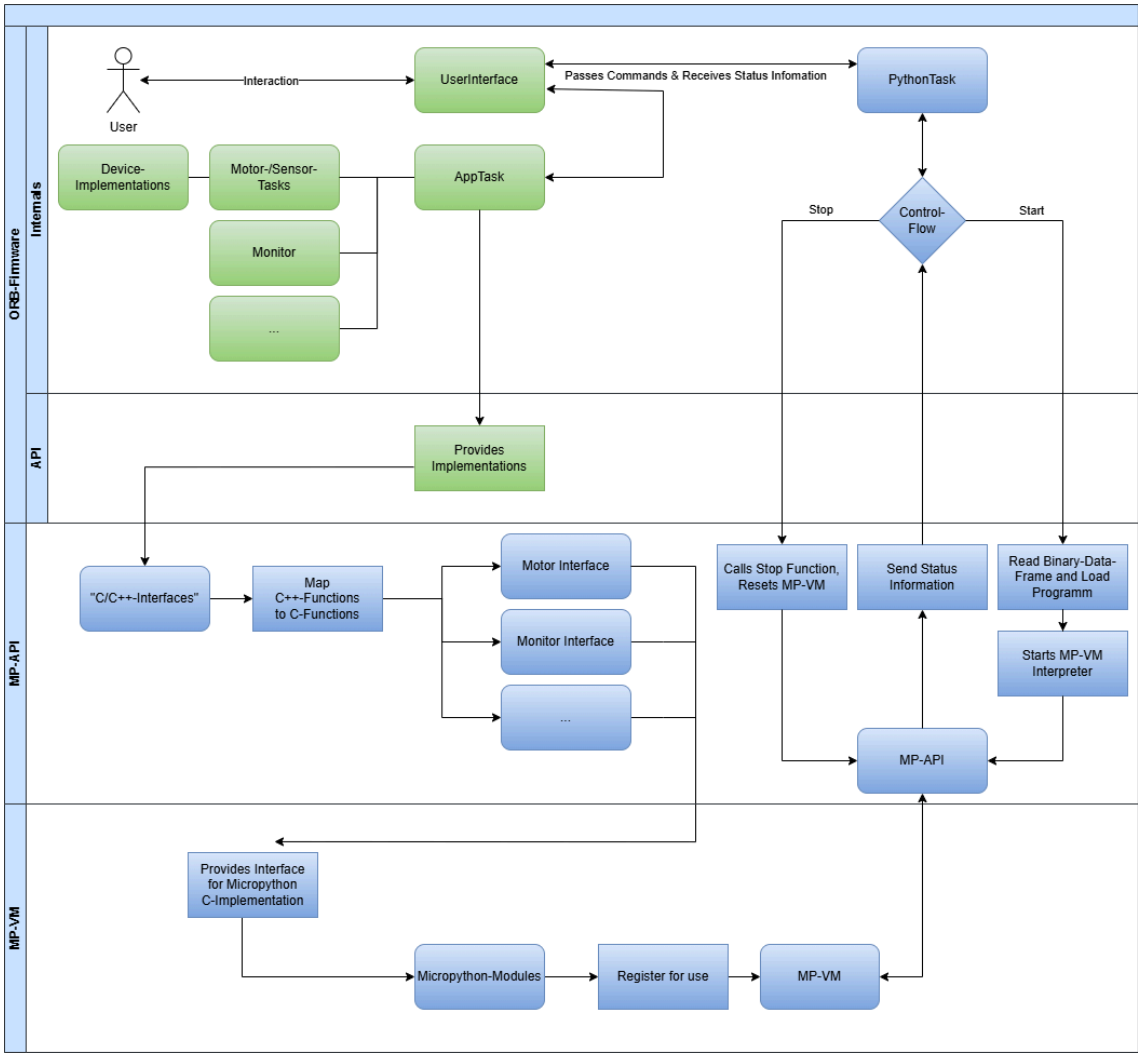

Abbildung 1: Firmware-Architektur

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.

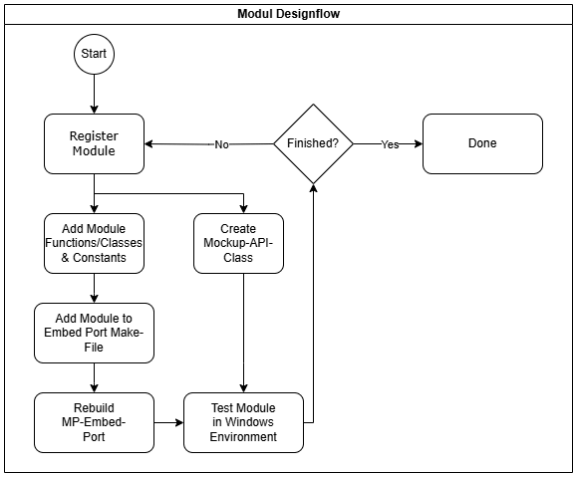

Abbildung 2: Module Designflow

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.

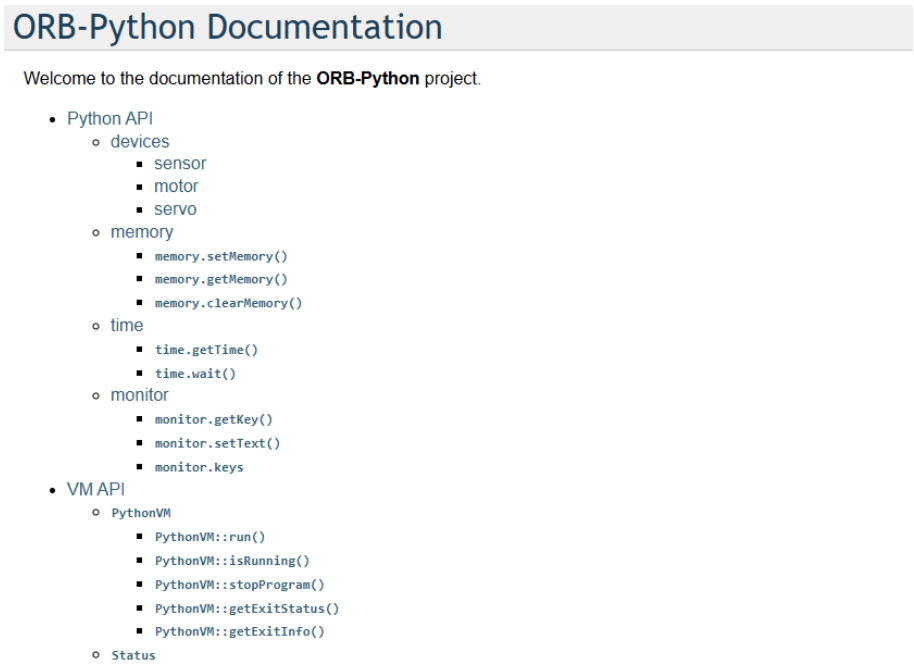

Abbildung 3: Python-API

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:

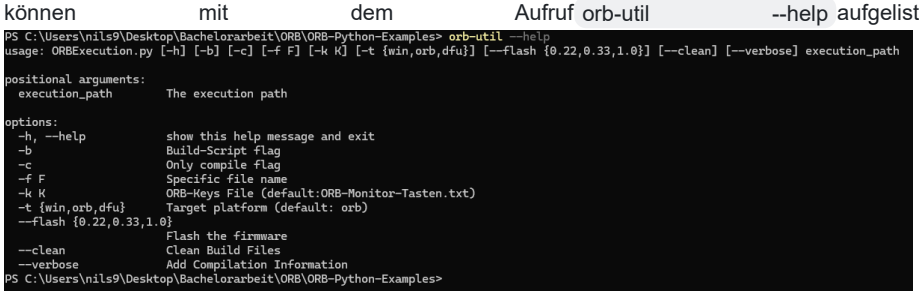

Abbildung 4: ORB-Util Help

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.

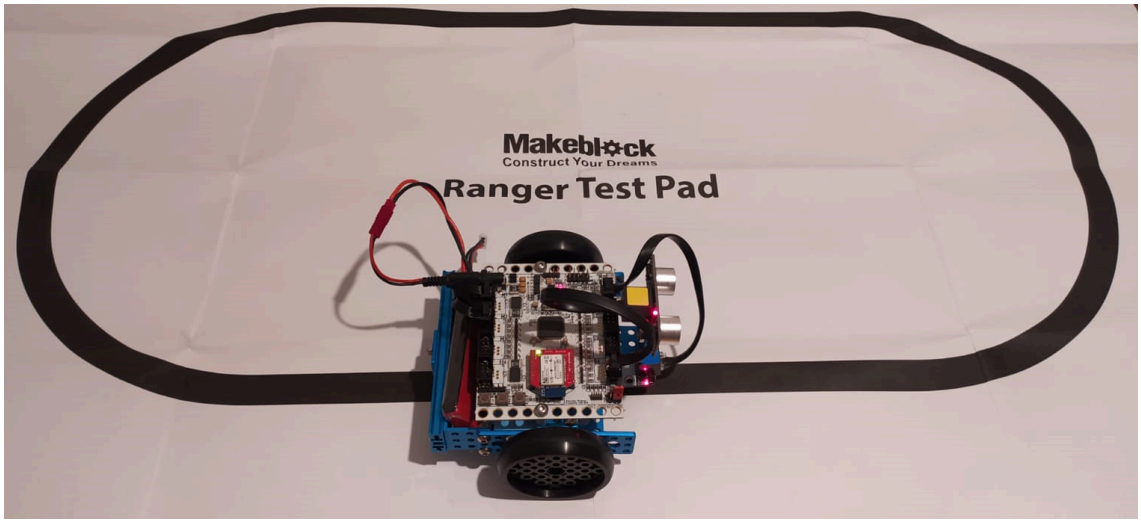

Abbildung 7: Strecke

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.