Konzepte

Inhaltsverzeichnis

- Warum den MicroPython Embed Port verwenden?
- MicroPython-Types
  - 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 Typ Zuordnung
  - Worauf stützt sich die Typ-Zuordnung
  - Wie funktioniert die Typ-Zuordnung (Nicht-Konkrete-Typen)
  - Wie funktioniert die Typ-Zuordnung (Konkrete-Typen)
- Problematik bei der Verwendung von Namespaces
  - Super- & Submodule
  - Limitierungen des MicroPython-Interpreters
  - Submodule als QString Alias
  - Kombination von Submodulen, QString und Supermodulen
  - Schlussfolgerung zu diesen Problemen
- MicroPython Flags
- Reduzierung des MicroPython-Heap-Verbrauchs durch Objektreferenzen
- Thread Safety
- Windows-Bug: Falsches Register bei Non-Local Return-Adressierung
  - Definition und Funktionsweise des Non-Local Return
  - Beschreibung des Bugs
  - Vorgehensweise zur Fehlerbehebung
- Compiler Flag Kompatibilität
  - MicroPython Compiler Flags
  - ORB-Firmware Compiler Flags
  - Angepasste Compiler Flags
- QStrings

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.