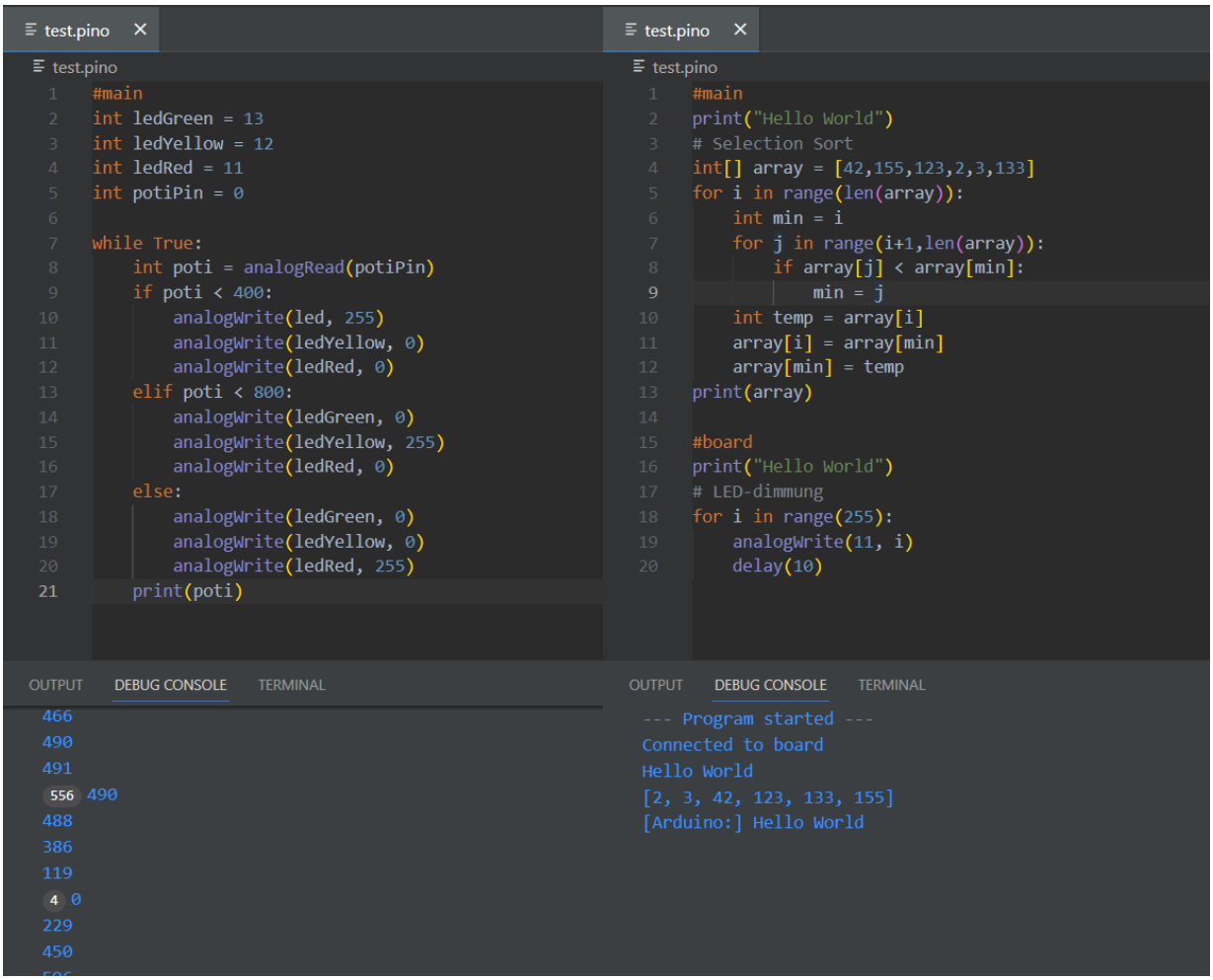
Pyduino

Christian Krause

SFZ Südwürttemberg Standort Ochsenhausen

Jugend Forscht 2023



Inhalt

Kurzfassung

In meinem Projekt habe ich eine einfachere Programmiersprache für den Microcontroller Arduino entwickelt. Sie soll mit einer Python-ähnlichen Syntax den Einstieg ins Programmieren erleichtern, da für den Arduino sonst die schwierigere Programmiersprache C verwendet werden muss. Um dies möglich zu machen, habe ich einen Transpiler in Python programmiert, der die Pyduino Syntax in C Syntax übersetzt und auf Fehler überprüft. Dieses C Programm wird dann für den PC und den Arduino kompiliert. So kann dieselbe Syntax auf beiden Plattformen ausgeführt werden. Wenn der PC mit dem Arduino verbunden ist, kann der PC auf die Ports des Arduino zugreifen und der Arduino kann Text in der Konsole des PCs ausgeben, auf Dateien zugreifen und Funktionen aufrufen. Mit Pyduino kann der Arduino vom PC aus gesteuert werden, ohne das Programm jedes Mal hochladen zu müssen und selbst auf die Rechenleistung eines PCs zugreifen, was die Einsatzmöglichkeiten des verbreiteten Microcontrollers stark erweitert.

Problem

Der Arduino ist ein sehr beliebter Microcontroller, mit dem sehr Viele die Grundlagen des Programmierens lernen. Im NWT-Unterricht in Baden-Württemberg, sowie in Werkrealschulen und Realschulen im Fach Technik ist er ein wichtiger Bestandteil [Q1]. Außerdem werden in vielen Bundesländern Lehrer für den Arduino fortgebildet. In einigen Hochschulen wird der Arduino zum Start in das technische Studium genutzt und er findet sich als Empfehlung sogar in der offiziellen Abschlussprüfung der IHKs [Q2]. Die Popularität des Arduino hängt damit zusammen, dass er einfach, benutzerfreundlich und universell einsetzbar ist. Auch der Einstieg ist sehr einfach: Innerhalb von wenigen Minuten kann selbst ein Anfänger die ersten Ergebnisse, wie zum Beispiel eine blinkende LED, sehen. Der Arduino ist aber trotzdem auch für fortgeschrittene Anwendungen geeignet. Dabei ist er aber immer noch sehr benutzerfreundlich, vor allem durch die neue Arduino IDE 2.x, die Features wie Syntax-Highlighting und Autovervollständigung für die Arduino-Programmiersprache zur Verfügung stellt. Diese Programmiersprache basiert auf C++ und ist mit einigen Hilfsfunktionen, mit denen zum Beispiel die Ein- und Ausgänge des Arduino angesteuert werden können, ausgestattet [Q3].

Da der Arduino oft von Programmieranfängern verwendet wird, um in das Programmieren einzusteigen, ist die Programmiersprache allerdings auch ein Problem. C++ ist eine low-level Programmiersprache, bei der dem Nutzer weniger „abgenommen“ wird als bei high-level Programmiersprachen, wie zum Beispiel Python. Das führt dazu, dass Programmieranfänger nicht nur grundlegende Programmierkonzepte, sondern auch spezielle Eigenheiten von C++ lernen müssen, die den Einstieg in die Informatik unnötig kompliziert machen.

In einer high-level Programmiersprache wie Python ist es zum Beispiel möglich, verkettete Listen, bei denen einfach Elemente angehängt und entfernt werden können, zu verwenden oder Funktionen zu programmieren, die solche Listen zurückgeben können. Außerdem sind viele hilfreiche Funktionen, wie zum Beispiel die **`len()`** Funktion oder die **`sum()`** Funktion, mit denen die Länge und Summe von Listen bestimmt werden können, in Python bereits enthalten.

Solche einfachen Dinge sind in C++ aber nur mit größerem Aufwand umsetzbar. Verkettete Listen sind in C++ nicht implementiert, d.h. entweder müssen sie selbst programmiert werden oder es müssen Arrays verwendet werden. Arrays können aber nur mit einer festen Größe und Datentyp definiert werden; es können also zum Beispiel keine Elemente eingefügt oder entfernt werden.

Für Anfänger wäre es daher deutlich komfortabler und intuitiver, mit Python in die Informatik einzusteigen. Aufgrund der vielen Vereinfachungen benötigt Python aber deutlich mehr Ressourcen und ist langsamer als C++. Deshalb ist es bisher nicht ohne weiteres möglich Python auf dem Arduino auszuführen.

Die Arduino IDE basiert auf dem Konzept, Programme zu kompilieren und auf den Arduino hochzuladen [Q4]. Diese Funktionen stellt das Arduino CLI zur Verfügung; sie sind aber sehr zeitaufwändig, was das Programmieren erschwert:

Plattform *(siehe Quellen)*	Kompilieren	Hochladen	Gesamt
PC	0.74s	1.85s	3.59s
Laptop	4.43s	2.78s	7.21s

(Die Zeiten wurden mit diesen Programmen ermittelt: https://github.com/Bergschaf/Arduino-Benchmark)

Außerdem konnte ich feststellen, dass die Zeit, die für das Kompilieren bzw. Hochladen benötigt wird, mit der Programmgröße nur leicht steigt, aber stark von der Leistungsfähigkeit des Systems abhängt. Da der Nutzer so jedes Mal bis zu einige Sekunden warten muss, nur um den Effekt einer kleinen Änderung im Programm zu sehen, sind diese Zeiten dennoch lang.

Dieses Problem lässt sich mit der bestehenden Methode aber nicht lösen, da die Arduino-Programme kompiliert und auf den Arduino Microcontroller hochgeladen werden müssen. Python-Programme werden dagegen von einem Interpreter ausgeführt, was fast ohne Zeitverzögerung funktioniert.

Lösungsidee

Um diese Probleme zu lösen, könnte man beim Einstieg ins Programmieren an den Schulen natürlich einfach mit einer high-level Programmiersprache wie zum Beispiel Python beginnen. Dabei bliebe aber ein großer Vorteil des Arduino auf der Strecke, der für den Siegeszug des Microcontrollers an den Schulen und in der Ausbildung ausschlaggebend war. Er bietet die Möglichkeit, nicht nur theoretisch zu programmieren, sondern auch mit

Elektronik zu experimentieren. Es können zum Beispiel Aktoren, wie LEDs, angesteuert und Sensoren ausgelesen werden. Es gibt zwar Ansätze diese zwei Aspekte der Elektronik und der einfachen Programmierung zusammenzuführen. Da Python aber deutlich mehr Ressourcen benötigt als C++ sind geeignete Kleinrechner wie der Raspberry Pi deutlich teurer als der Arduino und Microcontroller wie der ESP, der ebenfalls mit einer vereinfachten Python-Version programmiert werden kann, viel weniger etabliert. Durch die große Community und die zahlreichen, speziell auf den Arduino angepassten Sensoren kommen Anfänger am populären Microcontroller kaum vorbei.

Deshalb ist meine Idee, eine Programmiersprache zu entwickeln, deren Syntax von Python inspiriert, also möglichst einfach ist, die aber trotzdem so wenig Ressourcen verbraucht, dass sie auch auf dem Arduino ausgeführt werden kann. Um dies zu ermöglichen, müssen zwar einige Vereinfachungen gegenüber Python unternommen werden, die die Programmiererfahrung aber nicht signifikant einschränken. Diese Vereinfachungen machen es möglich, dass die Sprache von einem Transpiler in C++ Syntax übersetzt und danach für den Arduino kompiliert werden kann. Ein weiterer Vorteil der Sprache ist, dass sie nicht nur auf dem Arduino, sondern auch auf einem herkömmlichen Computer läuft. Programme können somit auf dem PC und auf dem Arduino parallel ausgeführt werden. Der Programmteil, der auf dem PC ausgeführt wird, hat dabei, sofern der Arduino verbunden ist, die Möglichkeit, die Pins auf dem Arduino anzusteuern. Der Programmteil auf dem Arduino kann über die Verbindung zum Beispiel Ausgaben an den PC senden, um sie dann in der Konsole anzuzeigen.

Diese Verbindung gibt dem Programmierer die Möglichkeit, den Arduino auch nur als Steuerungseinheit für die Pins zu verwenden und sie direkt vom PC aus zu steuern. Da so das Programm nicht jedes Mal von dem langsamen Arduino CLI kompiliert und hochgeladen werden muss, können so die Wartezeiten minimiert werden. Das Programm müsste nur von einem Transpiler in C++ übersetzt werden und dann von einem C++ Compiler für den PC kompiliert werden, was deutlich schneller als der Arduino Compiler ist. Weil mein Ansatz beide Welten von Arduino und Python verbindet, habe ich meine Programmiersprache „Pyduino“ genannt.

Pyduino soll mit einer IDE, die mit modernen Features wie Syntax-Highlighting, Autovervollständigung und automatischer Fehlererkennung eine möglichst gute Programmiererfahrung bieten. Da es sehr schwierig ist, eine eigene IDE zu entwickeln, war meine Idee, ein Plugin für die bekannte IDE Visual Studio Code von Microsoft zu entwickeln. Sie ist frei verfügbar und bietet eine gute API, die es ermöglicht, mit geringem Aufwand neue Programmiersprachen einzubinden.

Features

Programmstruktur

Pyduino-Programme sind in drei Teile aufgeteilt; Am Anfang des Programms können Funktionen definiert werden, die auf beiden Plattformen verfügbar sind. Der Programmteil hinter **`#main`** wird auf dem PC ausgeführt, der Programmteil hinter **`#board`** wird auf dem Arduino ausgeführt.

```
int x():
    return 1

#main
print("Hello world")

#board
print("Hello world")

# Das ist ein Kommentar
```

Kommentare werden mit **#** eingeleitet und werden vom Transpiler nicht berücksichtigt.

Variablen

Variablen werden mit dem Datentyp und einem Namen definiert und ihnen kann ein Wert zugewiesen werden.

```
int x = 42
float y = 2.0
float z = x / y
str s = "Hello World"
```

Arrays

Für alle dieser Datentypen können mit folgender Syntax auch Arrays erstellt werden.

```
int[] a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
float[] b = [3.14,4.2]
str[] c = ["hello", "world"]
```

Auf einzelne Elemente von Arrays kann mithilfe von Indices, die bei 0 beginnen, zugegriffen werden.

```
int[] array = [42, 15, 4, 8, 23, 16]
print(array[0]) # -> 42
print(array[3]) # -> 8
```

Funktionen

Funktionen werden in Pyduino mit dem Datentyp, dem Namen und den Paramtern in Klammern definiert. Wird die Funktion ohne Decorator definiert, dann wird sie auf der Plattform ausgeführt, auf der sie aufgerufen wird. Mit einem **`@main`** oder **`@board`** Decorator kann festgelegt werden, dass die Funktion immer auf der jeweiligen Plattform ausgeführt wird, egal von welcher Plattform die Funktion aufgerufen wird. Die Paramter und der Rückgabewert wird dabei ggf. über die Serielle Verbindung an die andere Plattform geschickt.

```
@main
float x(int a, float b):
    return a + b
```

Die Funktion **`x`** nimt einen **`int`** und einen **`float`** Wert als Paramter und gibt einen **`float`** Wert zurück. Da die Funktion mit einem **`@main`** Decorator definiert ist, wird sie immer auf dem PC ausgeführt, auch wenn sie auf dem Arduino aufgerufen wurde.

Builtins

print

Die **`print()`** Funktion gibt wie in Python Werte aus. Mehrere Werte können mit einem Komma getrennt werden. Bei Arrays und Listen werden die einzelnen Werte ausgegeben.

```
print("Hello World") # Hello World
print("Hello", "World") # Hello World
int[] a = [1, 2, 3]
print("Array", a) # Array [1, 2, 3]
```

Wenn diese Funktion auf dem Arduino ausgeführt wird und der Arduino mit dem PC verbunden ist, dann wird die Ausgabe in der Konsole auf dem PC angezeigt.

len

Mit der **`len()`** Funktion kann die Länge von Arrays bestimmt werden.

```
int[] array = [42, 15, 4, 8, 23, 16]
print(len(array)) # 6
```

delay

Die **`delay()`** Funktion hält das Programm für eine bestimmte Zeit in Millisekunden an. Dabei wird aber trotzdem auf Anweisungen von jeweils Arduino oder PC gewartet, die dann während dieser Zeit auch ausgeführt werden können.

```
delay(1000) # warte 1 Sekunde
```

analogWrite

Steuert die Pins am Arduino mit einem Wert von 0 - 255 an. Der Arduino, der eigentlich nur digitale Pins besitzt, kann die Spannung bestimmter Pins, die mit einer Welle markiert sind, durch Pulsweitenmodulation (PWM) steuern. Dadurch kann zum Beispiel die Helligkeit von LEDs gesteuert werden.

```
for i in range(255):
    analogWrite(11,i)
    delay(10)
```

Die LED an Port 11 wird langsam heller, egal ob das Programm auf dem PC oder auf dem Arduino ausgeführt wird.

analogRead

Liest die Spannung an den analogen Ports des Arduino in einem Wertebereich von 0 – 1023 aus.

```
print(analogRead(0))
```

Kontrollstrukturen

`if` Bedingungen, **`while`** Schleifen und **`for`** Schleifen funktionieren gleich wie in Python.

```
if i > 5:
    print("i ist größer als 5.")
elif i >= 0:
    print("i ist größer oder gleich wie 0")
else:
    print("i ist kleiner als 0")
```

Wenn die erste Bedingung wahr ist, dann wird der erste Teil hinter dem **`if`** Befehl ausgeführt. Wenn dies nicht der Fall ist, werden nacheinander die Bedingungen der **`elif`** Segmente überprüft. Wenn eine der Bedingungen wahr ist, wird der entsprechende Programmteil ausgeführt. Ansonsten wird der **`else`** Teil ausgeführt.

Eine **`while`** Schleife wird ausgeführt, solange eine Bedingung wahr ist.

```
int x = 0
while x < 10:
    x = x + 1
    print(x)
```

Diese Schleife gibt die Zahlen von 1 bis 10 aus.

Die **`for`** Schleife iteriert mit einer Zählvariable über einen Bereich von Zahlen, der mit **`range`** festgelegt wurde. Das grundlegende Schema sieht folgendermaßen aus:

```
for Zählvariable in range(start,end,step):
    #code
```

Beispiele:

```
for i in range(4,20,3):
    print(i)
```

Diese **`for`** Schleife gibt die Zahlen von 4 bis 20 in 3er Schritten aus.

`for` Schleifen können auch über Arrays iterieren.

```
int[] array = [42, 15, 314, 69, 20]
for i in array:
    print(i)
```

Diese **`for`** Schleife gibt die Elemente aus dem Array einzeln aus.

Umsetzung

Editor

Als Grundlage verwende ich die IDE Visual Studio Code [Q5]. Sie stellt eine API für Erweiterungen zur Verfügung, mit der auch Unterstützung für neue Programmiersprachen implementiert werden kann. Um erweiterte Funktionen für diese Sprachen zur Verfügung zu stellen, gibt es die Möglichkeit, einen Language Server mithilfe des Language Server Protocols (LSP) zu verwenden [Q6]. Die Idee hinter dem LSP ist, die Entwicklung von Features wie Autovervollständigung und Fehlererkennung für verschiedene Texteditoren so einfach wie möglich zu machen [Q7]. Deshalb wird die Implementierung dieser Features von den Texteditoren und IDEs getrennt, indem nur ein Language Server für jede Programmiersprache entwickelt werden muss. Dieser Language Server kann dann über das Language Server Protocol mit einem Texteditor kommunizieren und so Features wie Autovervollständigung, Fehlererkennung und Dokumentation zur Verfügung stellen. Das bringt auch den Vorteil, dass der Language Server in einer beliebigen Programmiersprache entwickelt werden kann, unabhängig von der API des Texteditors oder der IDE.

Daher war meine Idee, mithilfe der Bibliothek pygls [Q8] einen Language Server für Pyduino in Python programmieren. Die implementierten Features können dann mit einer Erweiterung in Visual Studio Code verwendet werden.

Um Pyduino Programme auszuführen, wird aber auch ein Transpiler benötigt, der den Pyduino-Code in C++ übersetzt. Der Code muss beim Übersetzen in C++ aber auch auf Fehler überprüft werden, da es sonst zu Compilerfehlern beim Compilieren des C++ Codes kommen kann. Diese Fehlermeldungen sind für Anfänger meist schwer zu verstehen, auch da sich der übersetzte C++ Code teilweise stark vom Original unterscheidet. Daher muss der Transpiler alle Fehler im Programm erkennen und konstruktive Fehlermeldungen erstellen. Eine

solche Fehlerüberprüfung wird auch beim Language Server benötigt. Deshalb habe ich mich entschieden, den Language Server und den Transpiler in einem Python Modul umzusetzen. Wenn ein Programm auf Fehler überprüft werden soll, wird es transpiliert und die Fehler werden gespeichert. Dabei ist der Transpiler so implementiert, dass er nicht bei einem Fehler den Transpiliervorgang abbricht. Stattdessen erkennt er, welche Art von Anweisung vorliegt, erstellt eine entsprechende Fehlermeldung und setzt das Transpilieren fort. Das ist wichtig, da der Language Server, während das Programm im Editor geschrieben wird, bei jeder Änderung die Fehler erkennen soll. Deshalb wird der Transpiler meist mit unvollständigen Programmen aufgerufen und muss trotzdem sinnvolle Fehlermeldungen erstellen. Die Daten, die dabei über das Programm gesammelt werden, könnten auch dafür verwendet werden, um Autovervollständigung zur Verfügung zu stellen.

Seit dem Regionalwettbewerb habe ich das Transpiler Modul neu geschrieben, da es die alte Architektur sehr schwierig gemacht hat, alte Features zu debuggen und neue hinzuzufügen. Die alte Aufbau ist in der Version der Langfassung, die beim Regionalwettbewerb abgegeben wurde, dokumentiert [Q9]. Die neue Version ist dagegen deutlich allgemeiner implementiert und macht es dadurch einfacher, neue Features hinzuzufügen.

Tokenizer

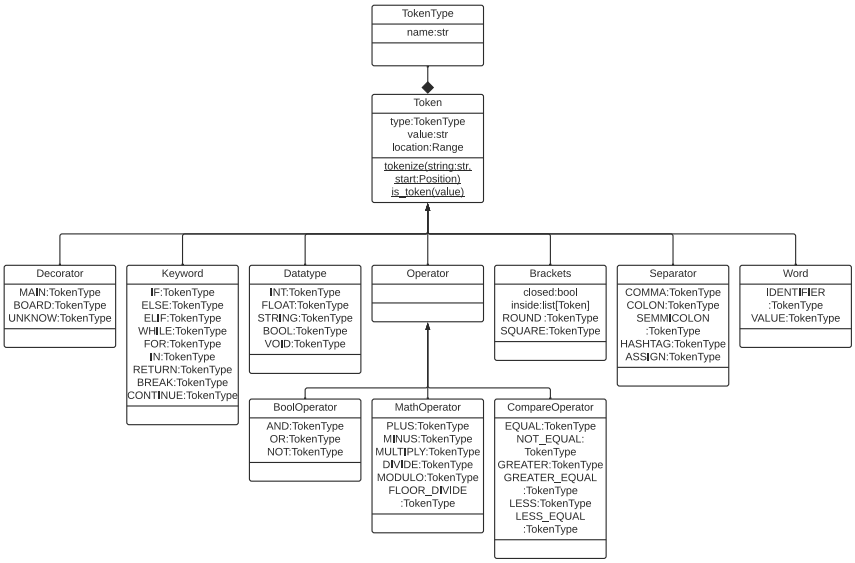


Abbildung 1: Klassendiagram für den Tokenizer

Um ein Pyduino Programm in Tokens umzuwandeln, wird die **`Token.tokenize()`** Funktion aufgerufen. Sie gibt eine Liste an Tokens (Instanzen der **`Token`** Klasse) zurück. In jedem Token ist die Position des Originalwertes im Pyduino Programm enthalten. Diese Position muss gespeichert werden, um in Fehlermeldungen genau angeben zu können, wo sich der entsprechende Programmteil befindet. Des Weiteren ist in jedem Token der Typ und ggf. der Wert des Tokens gespeichert.

Transpiler

Der Transpiler ist dafür zuständig ein Pyduino Programm anhand von den vorher generierten Tokens in C++ für den PC und für den Arduino zu übersetzen. Dabei soll er aber nicht nur korrekte Programme richtig übersetzen, sondern er soll auch unvollständige oder falsche Programme erkennen und konstruktive Fehlermeldungen ausgeben. Das Ziel ist dabei, dass der Transpiler alle Fehler abfängt, sodass es keine Fehlermeldungen vom Compiler gibt. Das ist wichtig, da diese für Anfänger oft sehr schwer lesbar sind, vor allem da der übersetzte Code, der vom Compiler kompiliert wird, oft stark vom Original abweicht. Da diese Fehlermeldungen aber auch direkt im Editor angezeigt werden sollen, darf der Transpiler nach einem Fehler nicht abbrechen. Er erstellt eine Fehlermeldung und überprüft das restliche Programm weiter auf Fehler.

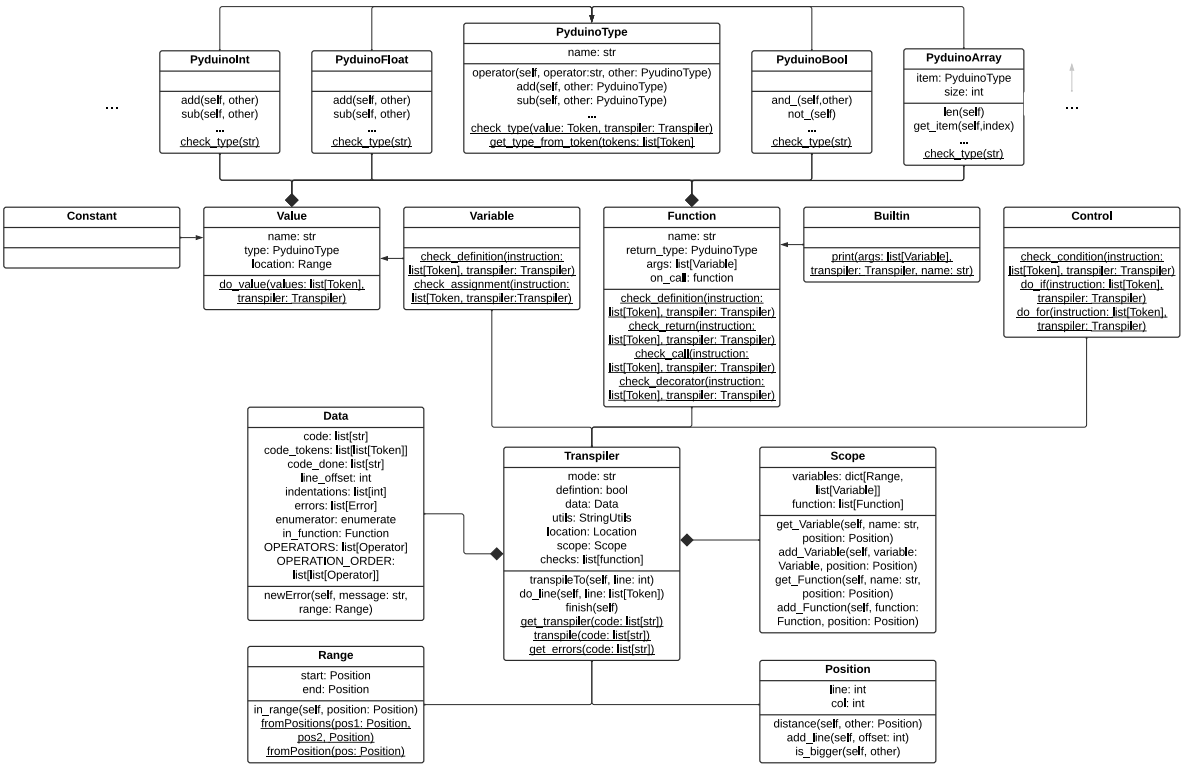


Abbildung 2: Klassendiagramm für den Transpiler

Die **`Transpiler`** Klasse ist für das Übersetzen der Tokens in C++ Code zuständig. Sie enthält ein **`mode`** Attribut, dass die Werte **`main`** und **`board`** annehmen kann und mit dem festgelegt wird, ob die jeweilige Instanz der Transpiler Klasse C++ Code für den PC oder für den Arduino generieren soll. Das ist wichtig, da sich bestimmte Aspekte beim Übersetzen in C++ zwischen den Plattformen unterscheiden. Die **`Transpiler`** Klasse wird jeweils einmal mit dem entsprechenden **`mode`** Attribut für den **`#main`** und für den **`#board`** Teil instanziiert. Für den Teil mit den Funktionsdefinitionen am Anfang des Programms wird die **`Transpiler`** Klasse jeweils einmal mit dem **`mode`** Attribut **`main`** und einmal mit dem **`mode`** Attribut **`board`** instanziiert. Das kommt daher, dass von beiden Plattformen auf die Funktionen zugegriffen werden soll.

Im gesamten **`Transpiler`** Modul werden die Klassen **`Range`** und **`Position`** verwendet, um die Positionen der Syntaxelemente im originalen Pyduino Code festzuhalten.

Die **`Transpiler`** Klasse enthält zudem die **`Data`** Klasse, in der generelle Informationen über das Programm gespeichert sind und die **`Scope`** Klasse, in der gespeichert wird, welche Variablen und Funktionen wo im Programm definiert sind.

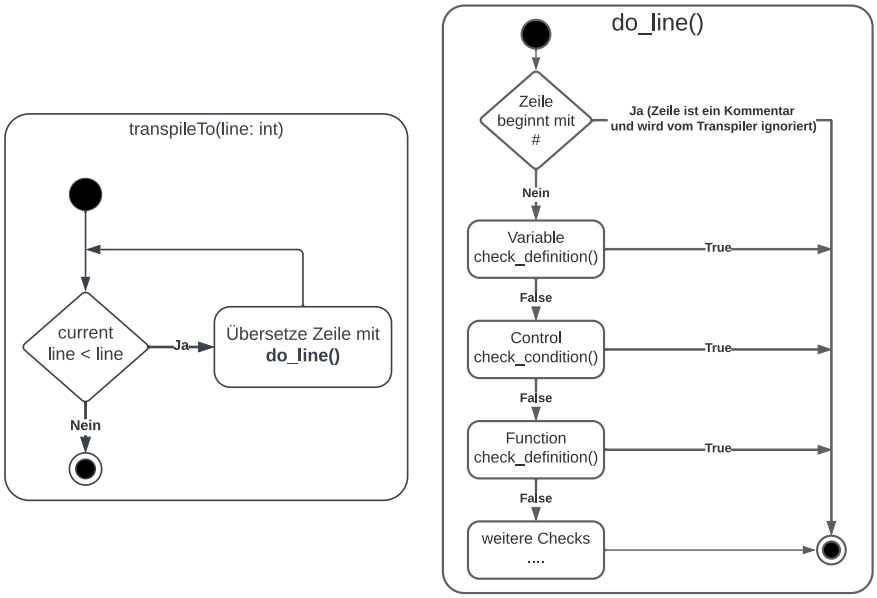


Abbildung 3: Ablaufdiagram für transpileTo() und do_line() Funktion

Um ein Pyduino Programm in C++ zu übersetzen, wird die **`Transpiler.transpileTo()`** Funktion verwendet. Diese übersetzt den Code mithilfe der **`Transpiler.do_line()`** Funktion zeilenweise bist zu der festgelegten Zeile. Die **`do_line()`** Funktion überprüft mit verschiedenen **`check`** Funktionen, welche Art von Anweisung in der Zeile vorliegt. Diese **`check`** Funktionen nehmen die Zeile als Liste aus Tokens und ein **`Transpiler`** Objekt als Paramter. Wenn die entsprechende Anweisung gefunden wurde, egal ob vollständig und valide oder nicht, wird **`True`** zurückgegeben, sonst **`False`**. Falls die gefundene Anweisung vollständig und valide ist, wird sie von der **`check`** Funktion in C++ übersetzt.

Zwei dieser **`check`** Funktionen sind zum Beispiel in der **`Variable`** Klasse implementiert. Die **`Variable.check_definition()`** überprüft, ob in der Zeile eine Variable definiert wird und die **`Variable.check_assignment()`** überprüft, ob in der Zeile einer Variable ein Wert zugewiesen wird. Wenn in Pyduino eine Variable definiert wird, dann wird beim Transpilieren für diese Variable ein **`Variable`** Objekt erstellt, dass den Namen, die Position und den Datentyp der Variable enthält.

Datentypen werden in Pyduino als Unterklassen der **`PyduinoType`** Klasse dargestellt. Die **`PyduinoType`** Klasse implementiert dabei für alle Operationen, die in Pyduino durchgeführt werden können, eine Funktion. Diese gibt zurück ob und wenn ja, wie diese Operation mit dem entsprechenden Datentyp durchgeführt werden kann. In der **`PyduinoType`** Oberklasse geben diese Funktion alle zurück, dass die entsprechende Operation nicht möglich ist. Die Klassen der verschiedenen Datentypen überschreiben die Funktionen für die Operationen, die mit dem entsprechenden Datentyp möglich sind mit Funktionen, die die entsprechende Operation durchführen und zurückgeben, dass die Operation möglich ist. Die **`PyduinoInt`** Klasse überschreibt dabei zum Beispiel die **`add()`** Funktion, die für das addieren zweier Zahlen zuständig ist. Diese Funktion nimmt einen anderen Datentyp als Argument und überprüft, ob sich ein **`int`** Wert mit dem anderen Datentyp addieren lässt. Dagegen wird die **`or()`** Funktion der **`PyduinoType`** Klasse in der **`PyduinoInt`** Klasse nicht überschrieben, da der **`or`** Operator in Pyduino nur für **`bool`** Werte verwendet werden kann. Wenn man also die **`add()`** Funktion eines **`PyduinoInt`** Objekts mit einem anderen **`PyduinoInt`** Objekt als Argument aufruft, wird zurückgegeben, dass die Operation möglich ist, indem man die Namen beider Werte links und rechts von einem Plus-Zeichen platziert. Wenn man dagegen die **`or()`** Funktion von einem **`PyduinoInt`** Objekt aufruft, wird mit der **`or()`** Funktion der **`PyduinoType`** Klasse zurückgegeben, dass die Operation nicht möglich ist.

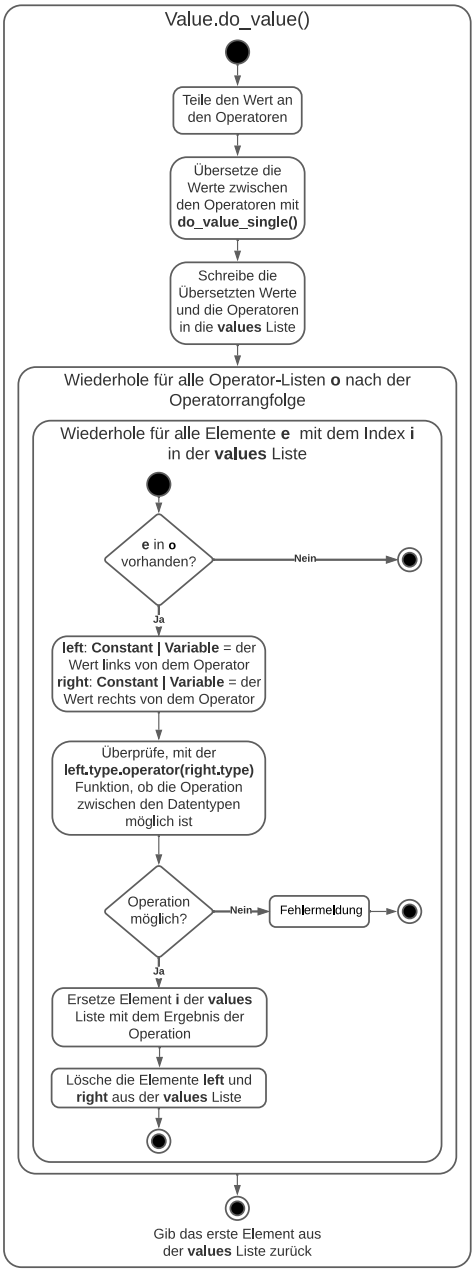
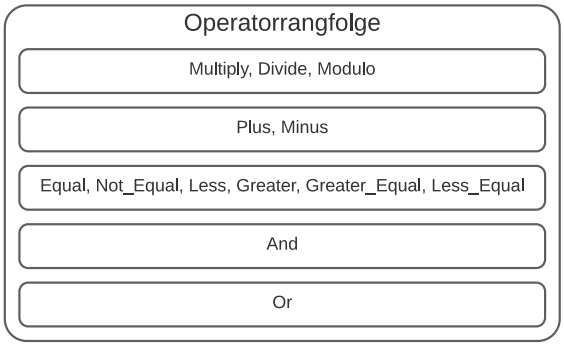


Abbildung 4: Ablaufdiagram für die do_value() Funktion

Eine weitere Funktion, bei der Datentypen sehr wichtig sind, ist die **`Value.do_value()`** Funktion. Sie nimmt einen Wert (eine Liste aus Tokens) und gibt ein **`Variable`** oder **`Constant`** Object zurück. Dabei überprüft sie rekursiv, ob alle Operationen mit den Datentypen innerhalb des Wertes möglich sind. Dafür wird der Wert zuerst nach den Operatoren aufgeteilt. Die Werte zwischen den Operatoren werden mit der **`Value.do_value_single()`** Funktion übersetzt. Diese Funktion überprüft zum Beispiel, ob es sich bei dem Wert um einen Variablennamen handelt uund ob diese Variable vorhanden ist. Wenn die **`Value.do_value_single()`** Funktion zum Beispiel mit einem **`BRACKETS.Round`** Token, der die Tokens innerhalb der Klammer enthält, aufgerufen wird, dann übersetzt sie die Tokens innerhalb der Klammer mit der **`Value.do_value()`** Funktion. So können auch verschachtelte Werte rekursiv überprüft und übersetzt werden.

Die übersetzten Werte (**`Constant`** oder **`Variable`** Objekte) werden zusammen mit den Operatoren in die **`values`** Liste geschrieben. Diese Liste wird dann nach der Operatorrangfolge übersetzt:



Dabei werden zuerst die Multiplikations, Divisions und Modulo Ausdrücke übersetzt. So werden Mathematische Regeln wie zum Beispiel „Punkt vor Strich“ eingehalten. Wenn alle Ausdrücke nach diesem Schema übersetzt sind, ist nur noch ein **`Constant`** oder **`Variable`** Objekt, das das Ergebniss des Ausdrucks darstellt, in der **`values`** Liste vorhanden. Dieses Element wird in der **`do_value()`** Funktion zurückgegeben.

In der **`Function`** Klasse sind weitere **`check`** Funktionen implementiert. Die **`Function.check_definition()`** überprüft zum Beispiel, ob in der Zeile eine Funktionsdefinition vorliegt und die **`Fuction.check_call()`** Funktion überprüft, ob in der Zeile eine Funktion aufgerufen wurde. Wenn eine Funktionsdefinition gefunden wurde, wird für die Funktion ein **`Function`** Objekt erstellt, das den Namen, die Position, den Datentyp des Rückgabewertes und die Datentypen der Parameter enthält. Falls vor der Funktion ein Decorator steht, der festlegt, auf welcher Plattform die Funktion aufgerufen wird, wird dieser in der **`Function.resolve_decorator()`** Funktion auf die Funktion angewendet. Wenn die Funktion auf der anderen Plattform ausgeführt werden soll, wird in der C++ Version der Funktion eine Anfrage an die andere Plattform über die serielle Verbindung geschickt.

Um einen Funktionsaufruf von Pyduino in C++ zu übersetzen, wird das **`on_call`** Attribut der **`Function`** Klasse verwendet. Es wird standartmäßig mit der Funktion **`standart_call()`** initialisiert. Diese Funktion übersetzt einen normalen Funktionsaufruf in C++, hinter den Funktionsnamen werden die Paramter in Klammern geschrieben.

Die **Builtin** Klasse, die von der **Function** Klasse erbt, ist für die standartmäßig in Pyduino vorhandenen Funktionen zuständig. Die Builtin Funktionen sind als Instanzen der **Buitlin** Klasse definiert. Manche Builtin Funktionen in Pyduino bestehen aber in der C++ Variante nicht nur aus einem herkömmlichen Funktionsaufruf, der mit der **standart_call()** Funktion übersetzt werden kann. Die **len(array)** Funktion in Pyduino wird zum Beispiel zu **(int)(sizeof(array) / sizeof(array[0])** übersetzt. Deshalb wird bei vielen Builtin Instanzen der **on_call** Paramter mit einer anderen Funktion als **standart_call()** initialisiert. Diese Funktionen sind als statische Funktionen der **Builtin** Klasse implementiert, zum Beispiel **Builtin.len()** oder **Builtin.delay()**.

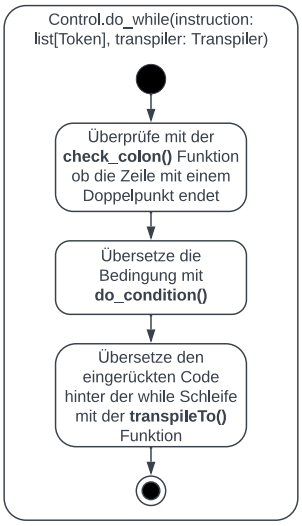


Abbildung 5: Ablaufdiagramm für die do_while() Funktion

Die **Control.check_condition()** Funktion überprüft, ob in der Zeile eine Kontrollstruktur, wie **if**, **while** oder **for** vorhanden ist. Wenn eine dieser Strukturen vorhanden ist, wird die entsprechende Funktion der **Control** Klasse (**do_if()**, **do_while()** oder **do_for()**) aufgerufen. Die **do_while()** Funktion überprüft zuerst, ob ein Doppelpunkt am Ende der Zeile vorhanden ist und übersetzt dann die Bedingung. Anschließend wird der eingerückte Programmteil nach der **while** Schleife mit der **transpileTo()** Funktion übersetzt. Diese ruft wiederrum mit der **do_line()** Funktion die verschiedenen **check** Funktionen auf. Da so auf den Code innerhalb einer Kontrollstruktur wieder die **Control.check_condition()** Funktion aufgerufen wird, werden so auch verschachtelte Strukturen rekursiv übersetzt.

Um ein Pyduino Programm auszuführen wird die **run()** Funktion der **Runner** Klasse aufgerufen. Sie ruft wiederum die **compile()** Funktion der Runner Klasse auf. Diese ist dafür zuständig, die Programme für den PC und für den Arduino zu kompilieren. Dafür ruft sie die **Transpiler.get_code()** Funktion auf, die ein Pyduino Programm als Parameter nimmt, für dieses **Transpiler** Objekte erstellt, den Code mithilfe der **transpileTo()** Funktionen übersetzt und den übersetzen Code für Arduino und PC zurückgibt. In der **compile()** Funktion wird dieser Code dann in temporäre Dateien geschrieben und dann von dem MinGW Compiler für den PC und vom Arduino ClI für den Arduino kompiliert. Anschließend wird überprüft, an welchem Port der Arduino angeschlossen ist, das kompilierte Programm wird mit dem Arduino ClI auf den Arduino hochgeladen und der PC Programmteil wird als **.exe** Datei ausgeführt.

Verbindung

Wenn die Programmteile parallel auf PC und Arduino laufen, haben sie die Möglichkeit, über die Serielle Verbindung Daten auszutauschen. Das folgende Diagramm zeigt die Funktionsweise dieser Verbindung;

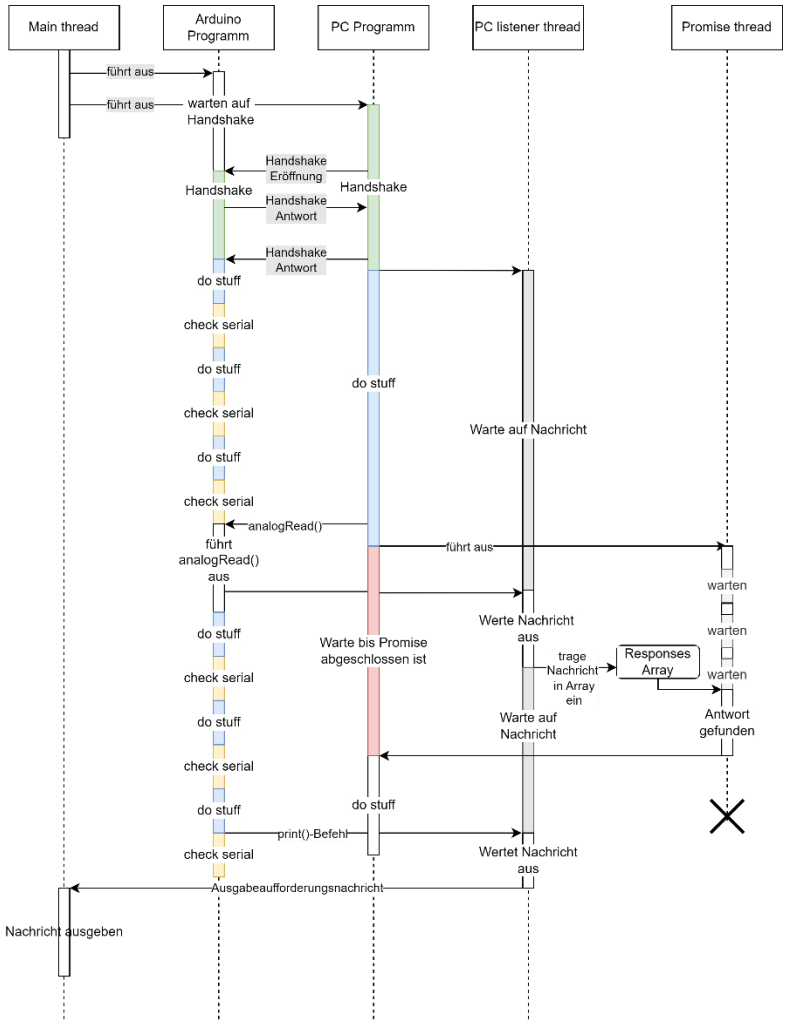


Abbildung 6: Verbindung zwischen PC und Arduino

Wenn ein PC- und ein Arduino-Programmteil vorhanden sind und die Verbindung benötigt wird, dann verbinden sich PC und Arduino zuerst über einen Handshake. Dabei werden vorher festgelegte Werte ausgetauscht, um die Verbindung zu überprüfen. Erst wenn der Handshake abgeschlossen ist, starten beide Programme. Dabei wird auf dem PC ein **`listener`** Thread gestartet, der auf Nachrichten des Arduino über die serielle Verbindung wartet. Da der Arduino kein Multithreading unterstützt, wird nach jedem Befehl überprüft, ob Daten im seriellen Buffer [Q10] vorhanden sind. Diese werden dann verarbeitet und ggf. werden die Befehle ausgeführt.

Wenn in dem PC-Programmteil ein Befehl auf dem Arduino ausgeführt werden soll, dann wird der Befehl über die serielle Verbindung an den Arduino geschickt. Dieser führt dann den Befehl aus und schickt die Antwort zurück an den PC. Sie wird dann von dem **`listener`** Thread ausgewertet und anhand der Nachrichten-ID in das **`Responses`** Array geschrieben. Um den Rückgabewert des Befehls zu erhalten, hat der PC-Programmteil einen neuen Promise, also eine Instanz der **`Promise`** Klasse, erstellt, nachdem er den **`analogRead()`** Befehl an den Arduino geschickt hat:

```
template<typename T>
class Promise {
public:
    typedef T(bytesToType)(char* bytes);
    thread *t;
    static void resolve(int requestID, bytesToType bytesToType, T* targetVariable, char
        Responses[MaxRequests][MaxDataLength]) {
        int millisecondsWaited = 0;
        while (Responses[requestID][0] == 0) {
            sleep_for(milliseconds(1));
            millisecondsWaited++;
            if (millisecondsWaited > MaxMillisecondsToWaitForData) {
                cout << "Error: Timeout while waiting for response" << endl;
                return;
            }
```

```
        }
        if(targetVariable == nullptr || bytesToType == nullptr) {
            Responses[requestID][0] = 0;
            return;
        }
        *targetVariable = bytesToType(Responses[requestID]);
        Responses[requestID][0] = 0;
    }

    Promise(T* targetVariable,bytesToType bytesToType, int requestID, char
        Responses[MaxRequests][MaxDataLength]) {
        // start a thread with the resolving function
        t = new thread(resolve, requestID, bytesToType, targetVariable, Responses);
    }

    // Destructor
    ~Promise() {
        // join the thread
        t->join();
    }
};
```

Die **`Promise`** Klasse wird mit einem Pointer zur Zielvariable, einer Funktion, die die Antwort zu dem entsprechenden Datentyp der Zielvariable konvertiert, einer Nachrichten ID und dem **`Responses`** Array initialisiert. Dabei wird ein neuer Thread mit der **`resolve()`** Funktion gestartet. Diese Funktion wartet, bis für die Nachricht mit der Nachrichten ID eine Antwort im **`Responses`** Array vorhanden ist, oder das Zeitlimit überschritten ist. Anschließend wird die Antwort zu dem entsprechenden Datentyp konvertiert und in die Zielvariable geschrieben. Um sicherzustellen, dass eine Antwort eingegangen ist und ein Wert in die Zielvariable geschrieben wurde, muss der Promise gelöscht werden, was den Destructor aufruft. In diesem wird gewartet, bis der Thread mit der **`resolve()`** Funktion abgeschlossen ist, was bedeutet, dass ein Wert eingegangen ist.

Um die Antwort auf eine synchrone Anfrage, bei der das Programm wartet, bis eine Antwort vom Arduino eingegangen ist, zu erhalten, wird der Promise direkt nach seiner Initialisierung wieder gelöscht, was den Destructor aufruft und den **`main`** Thread damit anhält, bis der Promise aufgelöst ist. Diese Methode wird auch in der zurzeit implementierten **`analogRead()`** Pyduino-Funktion angewendet. Das bedeutet, dass das Pyduino-Programm anhält, bis der Rückgabewert der **`analogRead()`** Funktion eingegangen ist.

Mit der Promise Klasse ist es aber auch möglich, asynchrone Anfragen, bei denen das Pyduino-Programm nicht angehalten wird, bis eine Antwort eingegangen ist, zu implementieren. Dabei würde der Promise gestartet werden, ohne ihn direkt wieder zu löschen. Da der Promise einen separaten Thread mit der **`resolve()`** Funktion startet, wird das Pyduino Programm nicht angehalten. Wenn der Rückgabewert dann gebraucht wird, kann der Promise gelöscht werden, was sicherstellt, dass der Wert vorhanden ist.

Für die Anfragen und Antworten zwischen PC und Arduino habe ich ein eigenes Protokoll entwickelt:

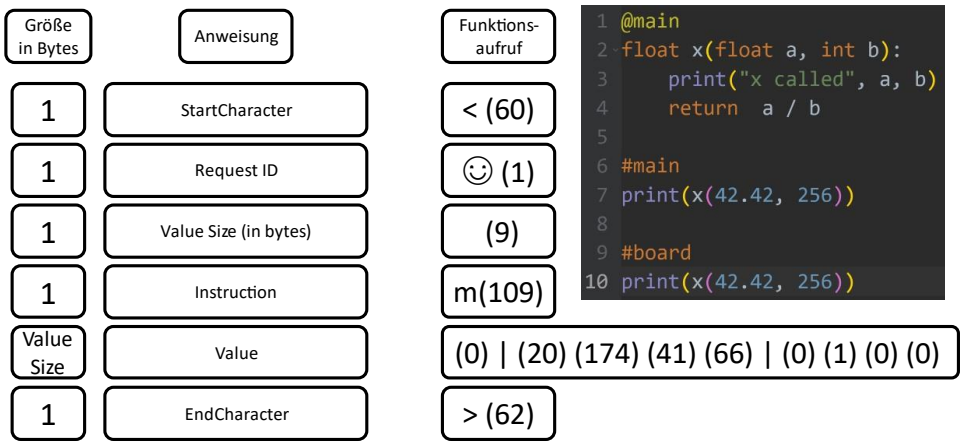


Abbildung 7: Serielle Anfragen

Eine Anfrage, die von einer Plattform zur anderen über die Serielle Verbindung geschickt wird, besteht aus einem Start Character, der den Anfang markiert und einer Request ID, die verwendet wird, um die Antworten auf die Anfragen zu unterscheiden. Danach folgt die Größe des Wertes, die Anweisung und ggf. der Wert. Am Ende wird die Anfrage von einem End Character abgeschlossen. In diesem Beispiel sieht man einen

Funktionsaufruf, der vom Arduino zum PC geschickt wird. Die Anweisung **109** bedeutet, dass es sich bei der Anfrage um einen Funktionsaufruf handelt. Das erste Byte repräsentiert die ID der Funktion, die Funktion **x** ist die erste Funktion in diesem Programm, also ist die ID **0**. Der erste Paramter der Funktion, **float a**, ist in den nächsten 4 Bytes codiert und der Paramter **int b** wird mit den letzten 4 bytes dargestellt. In der folgenden Abbildung ist die Antwort auf diese Anfrage dargestellt:

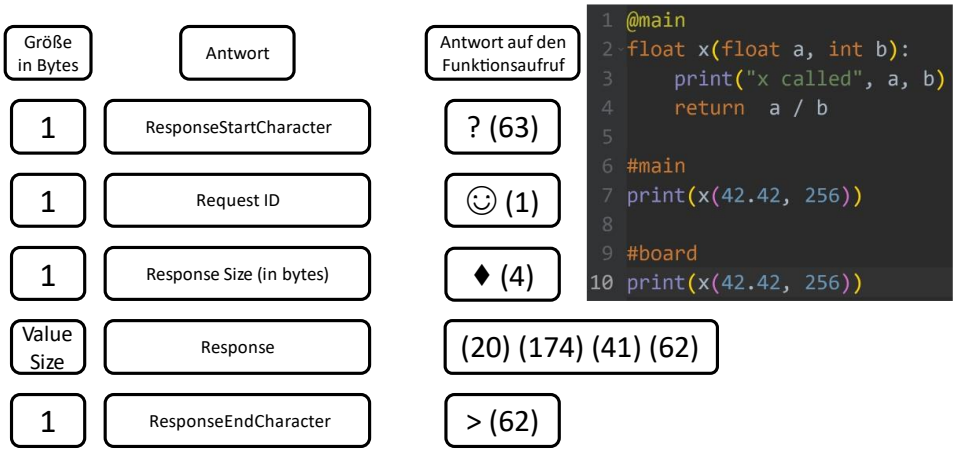


Abbildung 8: Serielle Antworten

Die Antwort wird von anderen Start und End Charactern begonnen und beendet, um sie einfach von einer Anfrage zu unterscheiden. Die Antwort enhält die Request ID der Anfrage, auf die sie antwortet. Dannach wird die Größe des Wertes und der Wert der Antwort übergeben. In diesem Beispiel gibt die Funktion einen **float** Wert zurück, der 4 Bytes entspricht.

VS Code Erweiterung

Dieses Diagramm zeigt die wichtigsten Dateien der VS Code Erweiterung, die auch aus dem Extension Marketplace heruntergeladen werden kann (https://github.com/Bergschaf/Pyduino-Plugin):

```
D:.
|   language-configuration.json
|   main.py
|   package-lock.json
|   package.json
```

In der **language-configuration.json** Datei befindet sich die grundlegende Konfiguration für die Pyduino-Sprache, die mit der Erweiterung als neue Programmiersprache registriert ist. Es wird zum Beispiel festgelegt, dass Kommentare mit **#** beginnen oder dass der Editor Klammern beim Tippen automatisch schließen soll.

Die **main.py** Datei kann in der Konsole mit dem Befehl **env/Scripts/python.exe main.py [Pyduino Datei]** aufgerufen werden, um eine Pyduino Datei zu kompilieren und auszuführen.

In der **package.json** Datei sind die Eckdaten der Erweiterung wie Name, Version und Veröffentlicher festgelegt. Außerdem ist beschrieben, dass die Erweiterung die Sprache „Pyduino“ zur Verfügung stellt und immer dann aktiviert wird, wenn eine **.pino** Datei geöffnet ist.

```
+---client
|   \---src
|       |   extension.ts
|
+---env
|   |   pyvenv.cfg
|   +---Include
|   +---Lib
|   +---Scripts
```

```
|   |   |    python.exe
```

Der **env** Ordner enthält die virtuelle Umgebung mit allen Bibliotheken, die benötigt werden, um den in Python implementierten Language Server zu starten. Dabei kann die **python.exe** Datei im **Scripts** Ordner verwendet werden, um Python Dateien auszuführen. Um die Größe der Datei kleiner zu halten, ist in der Erweiterung kein Python Interpreter enthalten. Daher muss dieser auf dem System installiert sein. Der Ort, an dem der Interpreter auf dem System zu finden ist, ist in der **pyvenv.cfg** Datei festgelegt. Da sich dieser Ort aber von System zu System unterscheidet, wird er bei jedem System neu bestimmt.

Die **activate()** Funktion der **extension.ts** Datei wird von VS Code aufgerufen, wenn die Erweiterung aktiviert wird, d.h. es wurde eine **.pino** Datei geöffnet. Sie ist dafür zuständig, den Language Server zu starten. Dafür wird die **python.exe** Datei in der virtuellen Python-Umgebung verwendet. Dafür muss aber zuerst der Pfad des Python Interpreters auf dem jeweiligen System bestimmt werden. Dieser wird dann in die **pyvenv.cfg** Datei geschrieben. Anschließend kann der Language Server ausgeführt werden.

```
+---mingw
|   |   MinGW.7z
|   \---MinGW
|       +---bin
|           c++.exe
|
+---node_modules
|
```

Da die Programme für den PC in C++ übersetzt werden, muss ein C++ Compiler verwendet werden, um die Programme zu ausführbaren **.exe** Dateien zu kompilieren. Da nur wenige Programmieranfänger einen C++ Compiler installiert haben, ist in der Extension der mingw C++ Compiler enthalten. Um die Dateigröße zu reduzieren, ist nur die **.7z** Datei enthalten, die bei Bedarf entpackt wird. In dem **mingw/MinGW/bin** Ordner ist die **c++.exe** Datei zu finden, mit der die C++ Programme kompiliert werden können.

```
+---server
|   |   server.py
|   +---transpiler
|
+---syntaxes
|       pyduino.tmLanguage.json
```

Der Language Server, der von der **extension.ts** Datei gestartet wird, ist in der **server.py** Datei implementiert. Bei jeder Änderung der **.pino** Datei wird die **did_change()** Funktion aufgerufen, die mithilfe der **Transpiler** Klasse das Programm auf Fehler überprüft.

In der **pyduino.tmLanguage.json** Datei sind die Regeln für das Syntax Highlighting, wenn eine Pyduino Datei in VS Code geöffnet ist, festgelegt.

Um Programme in VS Code auszuführen, können in der **.vscode/launch.json** Datei Run-Konfigurationen festgelegt werden. Daher wird von dem Language Server eine „Pyduino“ Konfiguration angelegt. Sie startet die **main.py** Datei mit der Python Installation im **env/Scripts** Ordner und der entsprechenden **.pino** Datei als Argument.

Getting Started

Um Pyduino Programme zu schreiben, kann die VS Code Erweiterung verwendet werden. Diese ist zurzeit nur mit Windows kompatibel. Auf dem System muss Python und 7zip installiert sein [Q11].

Als erstes muss VS Code installiert werden. Die Installationsdatei kann unter https://code.visualstudio.com/download heruntergeladen werden.

Danach kann die Pyduino-Erweiterung installiert werden. Dafür muss zuerst an der Seitenleiste der „Extensions“ Tab ausgewählt werden. Mit der Suchfunktion kann die Erweiterung unter „pyduino“ gefunden werden. Mit einem Klick auf „Install“ wird die aktuelle Version der Erweiterung installiert.

Um ein Pyduino-Programm zu schreiben, muss dann in VS Code eine Datei mit der Endung „.pino“ erstellt werden. Der Pyduino-Code wird dann automatisch auf Fehler überprüft, die dann rot unterstrichen werden, und es wird Syntax-Highlighting angewendet. Das Programm kann dann mit einem Klick auf den „Run Pyduino“ Knopf links unten auf dem Bildschirm ausgeführt werden.

Ausblick

Ein nächster Schritt für Pyduino ist es, neue Features zu implementieren. Dabei kommen zum Beispiel weitere Datentypen wie **`char`**, **`short`**, **`long`**, **`double`** und **`string`** in Frage. Für diese Datentypen können, wie in Python, dann auch verschiedene Builtin-Funktionen implementiert werden. Beispiele dafür wären **`string.find(value)`**, **`string.strip()`** oder **`array.sort()`**. Als weiteren Datentyp sind verkettete Listen geplant. Diese würden es, im Gegensatz zu Arrays, einfach machen, Elemente anzuhängen oder zu entfernen. Um Datentypen ineinander umzuwandeln, sollen auch Builtin-Funktionen wie zum Beispiel **`int(wert)`** oder **`string(wert)`** implementiert werden. Ein weiteres wichtiges Ziel sind Funktionen. Diese sollen, wie in Python, Argumente und optionale Keyword-Argumente erhalten und in der Lage sein, Werte zurückzugeben. Im Gegensatz zu C++ soll es auch ohne weiteres möglich sein, Datenstrukturen wie Arrays und Listen in Funktionen als Rückgabewert zu verwenden.

Der Austausch zwischen PC und Arduino könnte durch asynchrone Anfragen effizienter gemacht werden. Dabei würden Befehle oder Funktionsaufrufe an die jeweils andere Plattform geschickt, ohne auf die Antwort zu warten. Das Programm würde dann weiterlaufen, bis der Wert wirklich gebraucht wird. Erst dann würde die Antwort abgewartet werden, falls sie noch nicht da ist.

Ein wichtiger Vorteil des Arduino ist, dass viele Bibliotheken existieren, die in die Programme eingebunden werden können. Sie stellen zum Beispiel Funktionen für LCD-Displays oder spezielle Sensoren zur Verfügung. Da die Pyduino-Programme für den Arduino in C++ übersetzt werden, könnte auch Unterstützung für Bibliotheken, die in C++ geschrieben sind, eingebaut werden. Damit könnte man alle Arduino-Bibliotheken in Pyduino Programme einbinden und ihre Funktionen nutzen.

In der Zukunft könnte man sich auch damit beschäftigen, Unterstützung für Objektorientierung, also zum Beispiel Klassen und Vererbung, zu implementieren. Das hätte den Vorteil, dass es so möglich wäre, Objektorientierung direkt mit Pyduino zu lernen. Diese Fähigkeiten könnten dann reibungslos zu Python übertragen und dort weiter genutzt werden.

Um das Problem der Wartezeit, bis Arduino Programme kompiliert und hochgeladen sind, vollständig zu lösen könnte der Arduino beim Debugging auch nur als Steuereinheit für die Pins verwendet werden. Dabei würde am Anfang ein Programm auf den Arduino hochgeladen werden, in dem nur auf Befehle über die serielle Verbindung gewartet wird. Auf dem PC könnten dann Pyduino Programme ausgeführt werden, die auf den Arduino zugreifen, ohne jedes Mal beim Ausführen eines Programms einen Teil auf den Arduino hochladen zu müssen.

Es gibt aber Situationen, in denen das Hochladen der Programme auf den Arduino nicht zu vermeiden ist. Um die Wartezeit in solchen Situationen, in denen auch PC-Programme kompiliert werden müssen, möglichst zu reduzieren, wäre es möglich die Aufgaben mithilfe von Multiprocessing auf verschiedene Prozessorkerne zu verteilen und so parallel auszuführen.

Um das Programmieren mit Pyduino weiter zu erleichtern, wäre es möglich in dem VS Code Plugin neben Syntax Highlighting und Fehlererkennung Autovervollständigung zu implementieren. Das könnte mithilfe der Daten, die der Transpiler über das Programm sammelt, ebenfalls über den Language Server implementiert werden.

Um Pyduino robuster und weniger fehleranfällig zu machen, wäre eine Möglichkeit, mehr Unit Tests zu schreiben, die dann möglichst alle Features des Transpilers abdecken.

Außerdem besteht die Möglichkeit, die Pyduino VS Code Erweiterung als separate IDE zu veröffentlichen, die mit einer einzigen „.exe“ Datei installiert werden kann. Dafür würde die Theia Plattform [Q12] in Frage kommen, mit der auch die Arduino IDE 2.x implementiert ist. Das würde die Installation weiter vereinfachen.

Diese Weiterentwicklungen könnten das Potenzial der hier vorgestellten Programmiersprache Pyduino als einsteigerfreundliche Programmiersprache für den Arduino noch weiter ausbauen.

Dank

An dieser Stelle will ich Benno Hölz danken, die mir am SFZ immer meine Informatik-Fragen beantwortet haben und mich auch bei der Langfassung auf gute Ideen gebracht haben. Außerdem gilt mein Dank natürlich meinen Betreuern, Herrn Beck, dem Standortleiter des SFZ Ochsenhausen, und Herrn Ruf, die mich beide bei meinem Projekt begleitet haben.

Quellen

[Q1] https://www.schule-bw.de/service-und-tools/bildungsplaene/allgemein-bildende-schulen/bildungsplan-2016/beispielcurricula/gymnasium/BP2016BW_ALLG_GYM_NWT_BC_8-10_BSP_1.pdf

[Q2] https://www.ihk.de/blueprint/servlet/resource/blob/5556990/a9943739149bb694dba56dd7d272b678/h22-3280-b1-data.pdf

[Q3] https://en.wikipedia.org/wiki/Arduino#Sketch

[Q4] https://github.com/arduino/arduino-ide

[Q5] https://code.visualstudio.com

[Q6] https://code.visualstudio.com/api/language-extensions/overview

[Q7] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/

[Q8] https://github.com/openlawlibrary/pygls)

[Q9] https://github.com/Bergschaf/Pyduino/blob/76dab3e29f5931dea1f633007e96003e482c7591/doc/Langfassung.pdf

[Q10] https://www.arduino.cc/reference/en/language/functions/communication/serial/available/

[Q11] https://github.com/Bergschaf/Pyduino

[Q12] https://theia-ide.org

Abbildung 1-5: selbst erstellt mit Lucidchart (https://www.lucidchart.com/pages/de)

Abbildung 6: selbst erstellt mit Drawio (https://app.diagrams.net)

Abbildung 8, 9: selbst erstellt mit PowerPoint

PC: CPU: AMD Ryzen 5 5600X @3,7GHz 6-Cores 12-Threads; RAM: 16gb; OS: Windows 11

Laptop: CPU: Intel Pentium N3710 @1,6GHz 4-Cores 4-Threads; RAM: 4gb; OS: Windows 10