Femto

Eine Programmierumgebung für den ATmega-Mikrocontroller

Simeon Maxein

2. September 2010

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

Im Seminar Einchipcomputer [3] wurden Grundlagen für den Bau eines Computers gelegt, der im Wesentlichen aus einem Mikrocontroller der ATmega-Reihe von Atmel besteht. Sämtliche Peripheriefunktionen wie Tastatur- und Mausabfrage, Ansteuern von Massenspeichergeräten bis hin zur Ausgabe von Audio- und Videosignal sollten dabei durch geschickte Programmierung und kreative Nutzung der Hardware von diesem Chip erfüllt werden können.

Im Rahmen des Seminars wurden Lösungen für viele Teilbereiche entwickelt, beispielsweise für das Verwenden von PS2-Tastaturen und SD-Speicherkarten. Es wurden auch mehrere Anwendungen erstellt, die auf den Ergebnissen des Seminars aufbauen, zum Beispiel ein Sudokuspiel und eine Umsetzung von Sokoban.

Diese Anwendungen müssen jedoch auf einem gewöhnlichen PC entwickelt und dann auf den Mikrocontroller übertragen werden. Es wäre dagegen wünschenswert, Programme direkt auf dem Einchipcomputer erstellen und ausführen zu können, um einen flexibleren Einsatz zu ermöglichen.

Viele klassische Heimcomputer wie zum Beispiel der C64 von Commodore verfügen über einen integrierten Basic-Interpreter, der als Benutzeroberfläche dient und gleichzeitig eine Programmierumgebung zur Verfügung stellt. Das mitgelieferte Handbuch des C64, das keine Computerkenntnisse voraussetzt, erklärt sofort nach den Grundlagen der Bedienung des Gerätes das Programmieren in Basic, und da die Basic-Programmierumgebung sofort nach dem Einschalten des Geräts benutzt werden kann, ist die Hürde zum Erstellen erster kleiner Programme sehr gering. Da sich das gesamte Betriebssystem auf ROM-Chips befindet, und es deshalb nicht möglich ist, durch Programmierfehler dauerhaften Schaden an Rechner oder Betriebssystem anzurichten, kann man bedenkenlos experimentieren: Durch Aus- und Einschalten lässt sich der C64 innerhalb von Sekunden in seinen Ausgangszustand zurückversetzen. Diese Umstände machen den C64 zu einem idealen Computer für Programmieranfänger.

So entstand die Idee, für den Einchipcomputer eine Programmierumgebung mit Interpreter zu entwickeln, die Programmieranfängern ähnliche Vorteile bietet. Neben der Verwendung zu Lernzwecken gibt es auch weitere Einsatzmöglichkeiten: Wenn die interpretierte Sprache einen Zugriff auf die I/O-Funktionen des Mikrocontrollers ermöglicht, können innerhalb kurzer Zeit einfache Mess- und Regelanwendungen realisiert werden.

1.2. Zielsetzung

Wie oben motiviert ist das Ziel dieser Studienarbeit, eine Programmierumgebung zu entwickeln, die auf einem ATmega-Mikrocontroller von Atmel läuft. Dabei wird eine PS/2-Tastatur als Eingabegerät verwendet, sowie ein VGA-Monitor zur Ausgabe.

Anstelle von Basic soll jedoch eine strukturierte Programmiersprache mit C-ähnlicher Syntax interpretiert werden. So lässt sich das Gelernte leichter auf moderne Sprachen übertragen. Auch für den Einsatz zur Erstellung von Regelsoftware ist eine Sprache mit

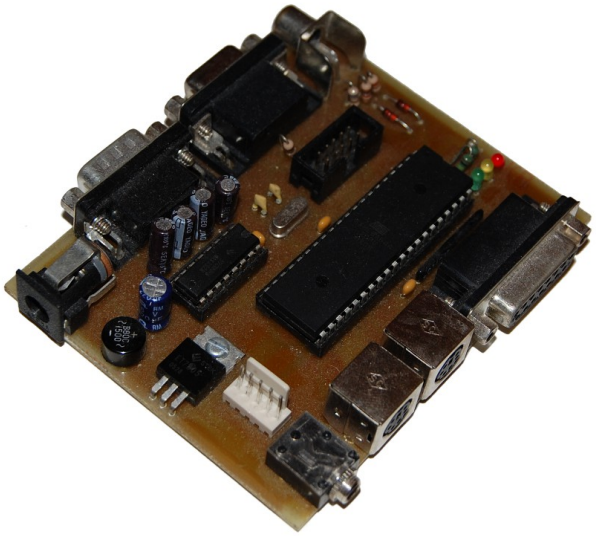

Abbildung 1: Die Einchipcomputer-Platine, die bei der Entwicklung verwendet wurde.

einer der C-Sprachfamilie entliehenen Syntax sinnvoll, da Entwickler in hardwarenahen Bereichen oft bereits mit der Entwicklung in C vertraut sind.

2. Übersicht über die Realisierung

2.1. Hardware

Als Hardwareplattform für die Umsetzung bot sich die Verwendung einer bestehenden Einchipcomputer-Platine an. Diese stellt die zum Betrieb des ATmega nötige Beschaltung zur Verfügung (Taktgenerator, Spannungsversorgung) und bietet Anschlüsse für verschiedene Ein- und Ausgabefunktionen, von denen in der hier vorgestellten Realisierung allerdings nur zwei genutzt werden: Eine 15-polige Mini-D-Sub-Buchse zur Ausgabe eines VGA-Videosignals und eine PS/2-Buchse zum Anschluss einer Tastatur.

Als Mikrocontroller wird ein ATmega644 eingesetzt, da dieses Modell über ausreichend viel Speicher verfügt, um die fertige Programmierumgebung sinnvoll einzusetzen. Insgesamt bietet der ATmega644 64K Bytes Flash-Speicher, 4K Bytes Arbeitsspeicher und 2K Bytes EEPROM [2, Seite 1]. Der Mikrocontroller folgt dabei der Harvard-Architektur: Der Flash-Speicher enthält den ausführbaren Programmcode, in diesem Fall von Interpreter und Programmierumgebung, während der Arbeitsspeicher die veränderlichen Daten bei der Programmausführung enthält. Der EEPROM-Speicher dient als zusätzlicher, nichtflüchtiger Speicher für selten veränderte Daten und kann in der Programmierumgebung zum Speichern eines Programms verwendet werden.

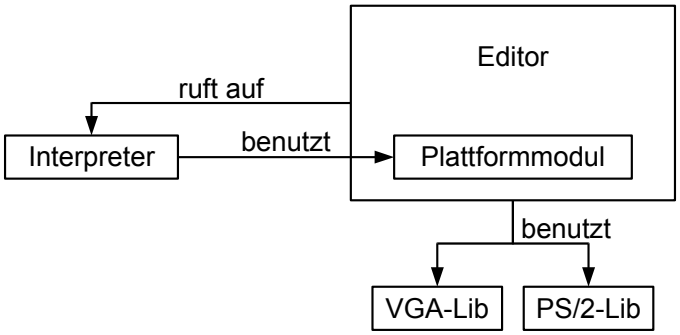

Abbildung 2: Zusammenspiel der Softwarekomponenten auf dem ATmega

2.2. Software

Die wichtigste Komponente der Programmierumgebung ist ein Interpreter mit dem Namen Femto, der eine selbst entwickelte JavaScript ähnliche Sprache interpretiert, die im weiteren Text als Femtoscript bezeichnet wird. Femto ist in C geschrieben und lässt sich mit wenig Aufwand auf andere Plattformen portieren.

Da der Interpreter portabel entwickelt wurde, muss er externe Funktionen und Makros für die Benutzung von plattformspezifischer Funktionalität wie Bildschirmausgabe und Tastaturabfrage aufrufen. Eine Implementierung dieser Funktionen wird hier als Plattformmodul bezeichnet. Um den Interpreter nutzen zu können, muss er gegen ein bestimmtes Plattformmodul kompiliert werden.

Neben dem Interpreter wurden im Rahmen dieser Arbeit zwei Bibliotheken für den ATmega erstellt, die auf den Lösungen aus dem Seminar Einchipcomputer aufbauen. Die erste Bibliothek ist für die Videoausgabe auf VGA-Monitoren vorgesehen und basiert auf der AVID-Bibliothek von Dr. Merten Joost, die zweite ist für das Abfragen einer PS2-Tastatur zuständig und hatte die Seminararbeit von Grégory Catellani [4] als Grundlage. Diese Bibliotheken implementieren die notwendige Funktionalität für wichtige Teile des ATmega-Plattformmoduls, stellen jedoch nicht exakt die vom Interpreter geforderten Headerdateien und Funktionen zur Verfügung, da sie auch unabhängig von Femto einsetzbar sein sollen und ansonsten zu eng mit diesem gekoppelt wären. Die Bibliotheken werden im Abschnitt 6 näher beschrieben.

Femto stellt keine Möglichkeit für das Bearbeiten von Quelltext zur Verfügung. Zu einer Programmierumgebung gehört also neben dem Interpreter wenigstens noch ein Texteditor, und eine Möglichkeit den Interpreter aufzurufen. Für diese Aufgaben wurde ein ATmega-spezifischer Editor entwickelt, der ebenfalls auf die beiden oben angesprochenen Bibliotheken zurückgreift. Neben dem Bearbeiten und Ausführen von Programmtext ist es auch möglich, diesen auf dem EEPROM des ATmega zu speichern und von dort wieder zu laden.

Taste	Funktion
Rücktaste	Zeichen links vom Cursor löschen
Entfernen	Zeichen rechts vom Cursor löschen
Pfeiltasten	Cursor in die entsprechende Richtung bewegen
Bild auf/ab	Cursor um mehrere Zeilen nach oben / unten bewegen
Pos1/Ende	Cursor an den Anfang / das Ende der Zeile bewegen
F2	Speichern ins EEPROM
F3	Laden von EEPROM
F5	Programm starten
F8	Textspeicher löschen
F12	Beispielprogramm laden

Tabelle 1: Übersicht der Tastenbelegung im Editor

Der Editor liefert darüber hinaus als Hauptanwendung eine main()-Funktion und enthält das vom Interpreter benötigte Plattformmodul, wobei für die eigentliche Ein- und Ausgabe wiederum auf die VGA- und PS/2-Bibliotheken zurückgegriffen wird.

3. Sprache und Programmierumgebung

3.1. Benutzung der Programmierumgebung

Direkt nach dem Einschalten des Einchipcomputers befindet man sich im Texteditor. Da der Textspeicher zunächst noch leer ist, ist nur ein leerer Bildschirm zu sehen. Mit der Tastatur kann man jetzt, wie aus Editorprogrammen am PC gewohnt, Text eingeben und bearbeiten. Zeilen, die über die Bildschirmbreite hinausgehen, werden dabei umgebrochen dargestellt. Fortgeschrittene Features wie Syntaxhervorhebung und automatische Einrückung werden nicht unterstützt.

Mit Hilfe der Funktionstasten kann man den aktuellen Textspeicher auf das EEPROM des ATmega sichern bzw. ein früher gespeichertes Programm wieder laden. Weiterhin lässt sich der Textspeicher löschen oder mit einem Beispielprogramm befüllen. Beim Laden und Löschen geht der aktuell im RAM befindliche Programmtext verloren, da er mit dem Inhalt des EEPROM überschrieben wird, ebenso überschreibt das Speichern ins EEPROM das vorher dort abgelegte Programm. Aus diesem Grund ist diesen Funktionen eine Sicherheitsabfrage vorgeschaltet, die ein versehentliches Überschreiben verhindern soll.

Die letzte Funktion ist das Ausführen des Programms im Textspeicher durch den Interpreter. Dabei wird zunächst der Bildschirm gelöscht und dann das Programm interpretiert. Die Programmausführung kann jederzeit durch Drücken der Tastenkombination Strg+c unterbrochen werden. Abschließend wird eine Meldung angezeigt, die darüber Aufschluss gibt, warum das Programm beendet wurde. Der Grund ist im Allgemeinen entweder das normale Ende der Ausführung durch das Programm selbst, ein Abbruch

auf Benutzerwunsch oder ein aufgetretener Fehler. Durch Drücken der Eingabetaste gelangt man zurück in den Editor. Um die Problemsuche zu vereinfachen, zeigt der Cursor im Texteditor nach dem Auftreten eines Fehlers auf die Programmstelle, an der die Programmausführung abgebrochen wurde. Bei einem Abbruch durch Drücken von Strg+c wird der Cursor ebenfalls an die Abbruchstelle versetzt, was zum Auffinden von Endlosschleifen hilfreich sein kann.

3.2. Aufbau eines Programms

Ein Femtoscript-Programm besteht grob betrachtet aus beliebig vielen Funktionsdefinitionen, gefolgt vom Hauptprogramm, bei dem die Ausführung beginnt. Ein einfaches Programm, das „Hallo Welt!“ auf dem Bildschirm ausgibt, sieht so aus:

```
1 print("Hallo Welt!");
```

print() ist dabei eine eingebaute Funktion von Femtoscript, die die Bildschirmausgabe realisiert. Die Klammern schließen die Parameterliste des Funktionsaufrufs ein, die in diesem Fall nur aus dem Stringliteral „Hallo Welt!“ besteht.

Das nächste, etwas anspruchsvollere Beispiel, berechnet rekursiv Zahlen der Fibonacci-Folge.

```
1  function fib(n) {
2      if(n<1) {
3          return 0;
4      }
5      if(n==1) {
6          return 1;
7      }
8      return fib(n-1)+fib(n-2);
9  }
10
11 var i;
12 print("Welche Fibonacci-Zahl berechnen?\n");
13 i = int(input());
14 print("Die Fibonacci-Zahl ", i, " ist ", fib(i));
```

Das Beispiel beginnt mit der Definition für die Funktion *fib()* (Zeilen 1-9), die einen Parameter n akzeptiert und die n-te Fibonacci-Zahl berechnen soll. Dazu werden mit Hilfe der beiden if-Blöcke die Ergebnisse für n=0 und n=1 vorgegeben, für alle größeren Zahlen wird das Ergebnis in Zeile 8 rekursiv berechnet.

In Zeile 11 beginnt das Hauptprogramm mit der Definition einer Variable i. Variablendefinitionen stehen immer am Anfang des Hauptprogramms (globale Variablen) oder der entsprechenden Funktion (lokale Variablen). In Zeile 13 wird mit Hilfe der eingebauten *input()*-Funktion eine Eingabe vom Benutzer angenommen. Die *input()*-Funktion gibt einen String zurück, der für die Fibonacci-Berechnungen zuerst in einen Integerwert um-

gewandelt werden muss. Das geschieht mit Hilfe der ebenfalls in Femtoscript eingebauten Funktion *int()*.

Zeile 14 demonstriert die Übergabe mehrerer Parameter an die print-Funktion. Diese Funktion übernimmt eine beliebige Zahl von Parametern[1] und gibt sie nacheinander auf dem Bildschirm aus. Außer *print()* besitzen nur wenige eingebaute Funktionen eine variable Parameterzahl, bei benutzerdefinierten Funktionen ist die Zahl der Parameter immer fest.

3.3. Typen

Femtoscript unterscheidet zwischen den vier Datentypen Integer (Ganzzahl), Float (Gleitkommazahl), String (Zeichenkette) und dem speziellen Typ Void (Nichts). Die Eigenschaften dieser Typen werden im Folgenden näher beschrieben.

3.3.1. Integer

Der Ganzzahltyp in Femtoscript entspricht dem C-Typ *int32_t*, wird also als 32-Bit Integer in Zweierkomplementnotation umgesetzt. Die kleinste darstellbare Zahl ist damit $-2^{31} = -2.147.483.648$, die größte $2^{31} - 1 = 2.147.483.647$.

Integerwerte werden auch benutzt, um logische Werte auszudrücken. Dabei steht Null für Falsch (false), jeder andere Wert für Wahr (true).

Integerliterale können dezimal, oktal oder hexadezimal angegeben werden. Falls das Literal mit 0x beginnt, werden die folgenden Zeichen als Hexadezimalzahl ausgewertet. Falls es nur mit 0 beginnt, werden die folgenden Zeichen als Oktalzahl interpretiert. Ansonsten wird das Literal als Dezimalzahl gelesen. Beispiele für gültige Literale sind 0x7e3b, 02511 und 42.

3.3.2. Float

Der Gleitkommatyp ermöglicht das Arbeiten mit gebrochenen Zahlen und wird im C-Quelltext des Interpreters als *float*-Wert behandelt. Die genaue Umsetzung dieses Typs hängt daher von C-Compiler ab, mit dem der Interpreter übersetzt wird, allerdings schreibt der C-Standard einen minimalen Wertebereich von -10^{37} bis 10^{37} und eine minimale Genauigkeit von sechs Dezimalstellen vor.

Gleitkommaliterale verwenden einen Punkt als Dezimaltrennzeichen und können einen Exponentialteil enthalten, der mit einem e oder E als Trennsymbol am Ende des Literals angefügt ist. Außerdem werden NAN und INF als die speziellen Gleitkommawerte „not a number“ und „infinity“ interpretiert. Beispiele für gültige Gleitkommaliterale sind 3.1415, 1.37e−20, 1e6, 22.0 und INF.

3.3.3. String

Ein String ist eine unveränderliche Zeichenkette, die bis zu 255 Zeichen lang ist und beliebige Zeichenwerte enthalten kann. Jede Operation, die einen String bearbeitet, er-

[1] Der Interpreter unterstützt allerdings maximal 255 Parameter für Funktionen.

Tabelle 2: Escape-Sequenzen für String-Literale

Zeichen	Escape-Sequenz	Kommentar
"	\"	Wird bereits als Stringbegrenzer verwendet
\	\\	Wird bereits als Escapezeichen verwendet
Zeilensprung	\n	Alternative Schreibweise, Zeilensprung in Stringliteralen ist ebenfalls erlaubt
Nullbyte	\0	Reserviert für die interne Begrenzung von Quelltextabschnitten im Interpreter

zeugt eine veränderte Kopie des Strings und belässt den ursprünglichen String in seinem Ausgangszustand. Strings folgen also wie die anderen Datentypen in Femtoscript einer Wertsemantik.

Stringliterale werden durch doppelte Anführungszeichen begrenzt und dürfen fast jedes Zeichen enthalten. Aufgrund der Arbeitsweise des Interpreters gibt es allerdings mehrere Zeichen, die durch Escape-Sequenzen ausgedrückt werden müssen. Diese Zeichen und ihre Escape-Sequenzen sind in Tabelle 2 aufgeführt. Abgesehen davon sind alle Zeichen erlaubt, allerdings kann man viele davon nicht auf der Tastatur eingeben. Um dennoch Strings zu erzeugen, die diese Zeichen enthalten, kann man die eingebaute Funktion *chr()* verwenden, die einen String aus einem Zeichen mit dem angegebenen Bytewert erzeugt.

Beispiele für gültige Stringliterale sind "Hallo \"Welt\"!", "" und "Neue\nZeile"

3.3.4. Void

Dieser spezielle Datentyp wird verwendet, wenn eine Variable keinen Wert enthält, und ist der Rückgabetyp von Funktionen, die keinen Wert zurückliefern. Das Verwenden eines Void-Wertes als Operand eines Ausdrucksoperators oder als Parameter einer eingebauten Funktion führt zu einem Typfehler und dadurch zu einem Programmabbruch.

Void dient also nicht zur Durchführung normaler Berechnungen, sondern in der Hauptsache dazu, dass Programmierfehler wie das Benutzen uninitialisierter Variablen nicht still ignoriert werden. Daher gibt es auch keine Möglichkeit, Void als Literal anzugeben.

3.4. Ausdrücke

Die Ausdrücke von Femtoscript orientieren sich stark an denen von C und Javascript. Operatoren mit zwei Operanden werden infix notiert, also zwischen ihre Operanden geschrieben. Die Auswertungsreihenfolge ist dabei durch die Prioritäten der Operatoren festgelegt, bei gleichen Prioritäten erfolgt die Auswertung immer von links nach rechts. Mit Hilfe von Klammerung kann die Auswertung von Teilausdrücken priorisiert werden.

Bei Operatoren mit zwei Operanden müssen beide Operanden den gleichen Datentyp besitzen. Wenn ein Operand ein Integerwert und der andere ein Floatwert ist, wird der

Integerwert allerdings automatisch in einen Floatwert umgewandelt bevor der Operator ausgewertet wird. Große Integerzahlen können bei dieser Umwandlung an Genauigkeit verlieren. Eine Umwandlung von und zu Stringwerten findet dagegen nicht automatisch statt; diese muss, falls erforderlich, mit Hilfe der eingebauten Funktionen *int()*, *float()* und *str()* durchgeführt werden. In der Regel hat der Ergebniswert eines Operators den gleichen Typ wie seine Operanden, Ausnahmen sind in der Beschreibung der einzelnen Operatoren vermerkt.

Als Operanden können in Ausdrücken neben Literalen auch Variablen und Funktionsaufrufe eingesetzt werden.

3.5. Bezeichner

Um auf Variablen und Funktionen Bezug zu nehmen, werden Bezeichner verwendet. In Femtoscript müssen Bezeichner mit einem Buchstaben beginnen, alle weiteren Zeichen können Buchstaben oder Ziffern sein. Die maximale Länge eines Bezeichners ist dabei 255 Zeichen.

Im Gegensatz zu vielen anderen Sprachen werden alle Zeichen zur Identifizierung der Variablen und Funktionen benutzt. Dadurch belegen lange Bezeichner allerdings mehr Hauptspeicher. Besonders bei rekursiven Funktionen ist es daher sinnvoll, kurze Namen für lokale Variablen und Parameter zu vergeben, um den Speicherbedarf bei der Ausführung gering zu halten.

Bei Bezeichnern (und auch bei Schlüsselwörtern) wird zwischen Groß- und Kleinschreibung unterschieden. radius, Radius, RADIUS und rAdIuS sind also vier vollkommen unterschiedliche Bezeichner.

Bezeichner für Funktionen und Variablen werden getrennt abgelegt, so dass eine Funktion den gleichen Namen besitzen darf wie eine Variable. Da Funktionsbezeichnern immer eine Parameterliste folgt, kann der Interpreter immer zwischen beiden Fällen unterscheiden.

3.6. Variablen

In einer Variable kann durch eine Zuweisung ein beliebiger Wert abgelegt werden. Femtoscript ist dynamisch typisiert, jede Variable kann also Werte jedes Typs aufnehmen.

Variablen können am Anfang des Hauptprogramms sowie am Anfang von Funktionen definiert werden. Dazu wird das Schlüsselwort **var** benutzt, gefolgt von einer kommaseparierten Liste der Variablenbezeichner, die mit einem Semikolon abgeschlossen ist.

Die Deklarationen können auch auf mehrere aufeinanderfolgende **var**-Listen aufgeteilt werden, allerdings müssen die Deklarationen vor allen anderen Anweisungen im Hauptprogramm bzw. in der Funktion erfolgen.

```
1 function calculate() {
2     var x,y,z;
3     var pi;
4 }
```

Tabelle 3: Liste aller Operatoren, die in Ausdrücken verwendet werden können. Die Liste ist sortiert nach absteigender Priorität der Operatoren; Operatoren gleicher Priorität sind zu Gruppen zusammengefasst. Die Spalte „Typen“ listet die Typen der Operanden auf, die für diesen Operator zulässig sind. Dabei werden die Typen Integer, Float und String mit ihren Anfangsbuchstaben abgekürzt.

Op.	Name	Typen	Beschreibung
−	Unäres Minus	i,f	Kehrt das Vorzeichen des nachfolgenden Operanden um.
~	Komplement	i	Kehrt jedes Bit des nachfolgenden Operanden um und berechnet so das Einerkomplement.
!	Logisches Nicht	i	Kehrt den logischen Wert des nachfolgenden Operanden um. Falls der Operand Null ist, ist das Ergebnis ungleich Null, sonst Null.
*	Multiplikation	i,f	Multipliziert die beiden Operanden.
:	Float-Division	i,f	Teilt den ersten Operanden durch den zweiten Operanden. Das Ergebnis ist immer ein Float-Wert, 5:2 hat daher als Ergebnis 2.5.
/	Division	i,f	Teilt den ersten Operanden durch den zweiten Operanden. Der Ergebnistyp entspricht dem Typ der Operanden, 5/2 hat daher als Ergebnis 2.
%	Modulo	i	Berechnet den Rest der Division des ersten Operanden durch den zweiten Operanden.
+	Addition / Konkatenation	i,f,s	Im Fall von Integer- und Floatoperanden werden die Operanden addiert, bei Strings wird eine Konkatenation durchgeführt.
−	Subtraktion	i,f	Subtrahiert den zweiten vom ersten Operanden.
<< >>	Bitweise Verschiebung	i	Verschiebt alle Bits im ersten Operanden um so viele Binärstellen nach links bzw. rechts, wie der zweite Operand vorgibt. Dabei wird mit Nullbits aufgefüllt.
< <= > >=	Ordnung	i,f	Prüft, ob die Aussage $l < r$, $l \leq r$, $l > r$ bzw. $l \geq r$ zutrifft, wobei l für den linken und r für den rechten Operanden steht. Falls die Aussage wahr ist, ist das Ergebnis ein Integerwert ungleich Null, sonst Null.
== !=	Vergleich	i,f,s	Prüft, ob die beiden Operanden den gleichen / einen unterschiedlichen Wert besitzen. Das Ergebnis ist wie oben ein Integer-Wahrheitswert.
&	Bitweises Und	i	Und-Verknüpfung der einzelnen Bits der Operanden.
^	Bitweises XOR	i	Exklusiv-Oder-Verknüpfung der einzelnen Bits der Operanden.
|	Bitweises Oder	i	Oder-Verknüpfung der einzelnen Bits der Operanden.
&&	Logisches Und	i	Falls beide Operanden ungleich Null sind, ist das Ergebnis ebenfalls ungleich Null, ansonsten Null.
||	Logisches Oder	i	Falls mindestens ein Operand ungleich Null ist, ist das Ergebnis ebenfalls ungleich Null, ansonsten Null.

Es ist auch möglich, Variablen direkt bei der Deklaration einen Wert zuzuweisen. Dazu schreibt man in der Deklaration hinter den Variablenbezeichner ein Gleichheitszeichen, gefolgt von einem Ausdruck, dessen Wert an die Variable zugewiesen werden soll. Der Ausdruck darf sich allerdings nicht auf Variablen beziehen, die erst später deklariert werden.

```
1 function circumference(radius) {
2     var diameter = radius*2, pi=3.1415;
3     return diameter*pi;
4 }
```

Falls die Variable nicht auf diese Art initialisiert wird, enthält sie nach der Deklaration zunächst den Typ Void. Wenn man Variablen in einem Ausdruck benutzt, ohne ihnen vorher einen Wert zuzuweisen, erhält man daher einen Typfehler.

Variablen, die im Hauptprogramm definiert werden, sind globale Variablen, auf die an jeder Stelle im Programm zugegriffen werden kann, insbesondere auch aus Funktionen heraus. In Funktionen definierte Variablen sind dagegen lokal und können nur innerhalb der Funktion benutzt werden, in der sie definiert sind. Jeder Funktionsaufruf reserviert Speicher für seine lokalen Variablen, so dass beim rekursiven Aufruf einer Funktion alle Ebenen der Ausführung mit getrennten lokalen Variablen arbeiten.

Eine lokale Variable versteckt eine globale Variable, wenn beide gleich bezeichnet sind. Jedes Vorkommen des Bezeichners innerhalb der betroffenen Funktion bezieht sich dann auf die lokale Variable. Es existiert keine Möglichkeit, explizit die globale Variable anzusprechen.

3.7. Funktionen

Femtoscript unterstützt Unterprogramme in Form von Funktionen, die Parameter entgegennehmen und einen Ergebniswert zurückgeben.

Funktionen müssen vor Beginn des Hauptprogramms definiert werden. Dazu wird das Schlüsselwort **function** verwendet, gefolgt vom Bezeichner der Funktion. Darauf folgt in runden Klammern die Parameterliste, hinter der sich in geschweiften Klammern der eigentliche Funktionskörper anschließt. Die Parameterliste besteht aus den Namen der Parameter, getrennt durch Kommata. Der Funktionskörper beginnt (optional) mit Variablendeklarationen, gefolgt von einer beliebigen Zahl von Anweisungen (statements).

Eine spezielle Anweisung, die direkt mit dem Funktionsablauf verbunden ist und deshalb hier erläutert wird, ist die **return**-Anweisung. Diese beendet die Abarbeitung der Funktion und legt den Rückgabewert fest. Zu diesem Zweck kann hinter dem Schlüsselwort **return** ein Ausdruck folgen, dessen Ergebnis als Rückgabewert benutzt wird. Falls kein Ausdruck angegeben wird, ist der Rückgabetyp Void. In jedem Fall muss die **return**-Anweisung mit einem Semikolon abgeschlossen werden.

Es ist auch zulässig, keine **return**-Anweisung in einer Funktion zu benutzen. Wenn die Programmausführung das Ende einer Funktion erreicht, wird automatisch Void zurückgegeben.

```
1 function max(a,b) {
2     if(a>b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
```

Funktionen werden normalerweise in Ausdrücken aufgerufen. Wenn die Funktion keinen Wert zurückgibt oder der Rückgabewert nicht benötigt wird, kann ein Funktionsaufruf aber auch als eigene Anweisung verwendet werden, die durch ein Semikolon abgeschlossen wird. In beiden Fällen besteht ein Funktionsaufruf aus dem Namen der Funktion, gefolgt von der Liste der Aufrufparameter in runden Klammern. Die Anzahl der Aufrufparameter muss der Zahl der Parameter in der Funktionsdefinition entsprechen.

Jeder Aufrufparameter ist ein Ausdruck, der vor dem Aufruf der Funktion ausgewertet wird. Die Parameter werden immer nach einer Wertsemantik übergeben (call by value), der Wert eines Parameters wird also immer in eine implizit definierte lokale Variable der aufgerufenen Funktion kopiert. Daher ist es zum Beispiel nicht möglich, eine Funktion wie *swap(x,y)* zu schreiben, die die übergebenen Parameter vertauscht. Während der Funktionsausführung werden die Parameter wie lokale Variablen behandelt, denen bereits zu Beginn der Funktion die Werte der Aufrufparameter zugewiesen wurden.

Das folgende Beispiel zeigt beide Arten von Funktionsaufrufen. Die eingebaute Funktion *print()* wird in Form einer eigenen Anweisung aufgerufen, während im Aufrufparameter die oben definierte Funktion *max()* als Teil eines Ausdrucks benutzt wird.

```
1 print(max(x,y+1)*2);
```

3.8. Hauptprogramm

Das Hauptprogramm folgt auf die Funktionsdefinitionen und ist der Teil des Quelltextes, bei dem der Interpreter die Ausführung startet. Es hat den gleichen Aufbau wie ein Funktionskörper. Zuerst werden die (globalen) Variablen deklariert, darauf folgen beliebig viele Anweisungen.

Das Ausführen einer **return**-Anweisung im Hauptprogramm beendet die Programmausführung. Falls ein Ausdruck als Rückgabewert angegeben ist, wird dieser Ausdruck noch ausgewertet, der errechnete Wert wird aber nicht weiter verwendet. Die Programmausführung endet auch, wenn das Ende des Programmquelltextes erreicht ist.

3.9. Anweisungen

Abgesehen von den bereits eingeführten Anweisungen (Funktionsaufruf und **return**-Anweisung) existieren noch weitere Arten von Anweisungen, die in den nächsten Abschnitten erläutert werden.

3.9.1. Zuweisungen

Diese Art von Anweisung wird benutzt, um einer Variable einen Wert zuzuweisen. Dazu wird zuerst der Name der Variable geschrieben, gefolgt von einem Gleichheitszeichen und einem Ausdruck. Der Ausdruck wird ausgewertet, und das Ergebnis wird der Variable zugewiesen.

Im Gegensatz zum gleichen Sprachkonstrukt in C ist eine Zuweisung in Femtoscript selbst kein Ausdruck, Konstruktionen wie a=b=c sind also nicht zulässig.

```
1 var pi;
2 pi = 3.141592;
```

3.9.2. if-else-Bedingungen

Die if-else-Anweisung ermöglicht die bedingte Ausführung von Programmcode. Auf das Schlüsselwort **if** folgt in runden Klammern ein Ausdruck, der beim Erreichen des if-Statements ausgewertet wird. Das Ergebnis muss ein Integerwert sein, der als Wahrheitswert interpretiert wird. Hinter dem geklammerten Ausdruck folgt ein Block aus beliebig vielen Anweisungen, der in geschweifte Klammern eingefasst ist und weiter unten als if-Block bezeichnet wird. Im Gegensatz zu vielen Programmiersprachen mit ähnlicher Syntax müssen diese geschweiften Klammern in Femtoscript immer geschrieben werden, auch wenn sich nur eine einzige Anweisung im Block befindet.

Nach der schließenden geschweiften Klammer des if-Blocks kann das Schlüsselwort **else** angegeben werden. In diesem Fall folgt dem **else** ein weiterer Anweisungsblock, der wieder durch geschweifte Klammern begrenzt sein muss und hier als else-Block bezeichnet wird.

Falls der oben besprochene Ausdruck als Wahr ausgewertet wird, also zu einem Integerwert ungleich Null, wird anschließend der if-Block ausgeführt, und der else-Block, falls dieser existiert, wird übersprungen. Falls der Ausdruck als Falsch (Null) ausgewertet wird, wird statt dessen der if-Block übersprungen und der else-Block wird ausgeführt, falls er existiert.

```
1 if(x%2 == 0) {
2     print("x ist gerade.");
3 } else {
4     print("x ist ungerade.");
5 }
```

3.9.3. while-Schleifen

Um einen Codeabschnitt wiederholt auszuführen, gibt es drei Arten von Schleifen in Femtoscript. Die while-Schleife beginnt mit dem Schlüsselwort **while**, dem wie der if-Anweisung ein geklammerter Ausdruck (die Laufbedingung) und ein Anweisungsblock in geschweiften Klammern (der Schleifenkörper) folgen. Falls die Laufbedingung zu Wahr

ausgewertet wird, wird der Schleifenkörper ausgeführt, ansonsten wird er übersprungen und die Ausführung wird hinter der Schleife fortgesetzt.

Nach der Ausführung des Schleifenkörpers wird die Laufbedingung erneut ausgewertet. Falls sie wieder zu Wahr ausgewertet wird, wird der Schleifenkörper erneut ausgeführt. Dieser Vorgang wird wiederholt, bis die Laufbedingung einmal zu Falsch ausgewertet wird, was zum Abbruch der Schleife führt.

Der folgende Quelltext gibt so lange eine zufällige Folge von Einsen und Nullen auf dem Bildschirm aus, bis der Benutzer eine beliebige Taste drückt. Falls der Benutzer schon zu Beginn der Schleife eine Taste gedrückt hat, wird nichts ausgegeben.

```
1 while(len(get())==0) {
2     print(rndch(0,1));
3 }
```

3.9.4. do-while-Schleifen

Diese Schleife ähnelt der while-Schleife, allerdings wird die Laufbedingung erst nach dem ersten Schleifendurchlauf das erste Mal überprüft. Der Schleifenkörper wird also mindestens einmal ausgeführt.

Diese Eigenschaft wird auch durch die Syntax der do-while-Schleife angedeutet. Die Schleife beginnt mit dem Schlüsselwort **do**, direkt gefolgt vom Schleifenkörper in geschweiften Klammern. Erst dahinter folgt das Schlüsselwort **while** und die Laufbedingung. Die do-while-Schleife wird durch ein Semikolon abgeschlossen.

Das folgende Beispiel entspricht dem für die while-Schleife, allerdings wird mindestens ein Zeichen ausgegeben, auch wenn der Benutzer schon vor Betreten der Schleife eine Taste gedrückt hat.

```
1 do {
2     print(rndch(0,1));
3 } while(len(get())==0);
```

3.9.5. for-Schleifen

Die for-Schleife besteht wie die while-Schleife aus einem Schleifenkopf und einem Schleifenkörper, allerdings enthält der Schleifenkopf im Fall der for-Schleife neben der Laufbedingung noch zwei Zuweisungen. In den runden Klammern nach dem for-Schlüsselwort ist zuerst die erste Zuweisung angegeben, dann folgt ein Semikolon und die Laufbedingung, ein weiteres Semikolon und schließlich die zweite Zuweisung.

Die erste Zuweisung wird einmal zu Beginn der Schleife ausgeführt, wenn die Programmausführung die for-Anweisung erreicht. Diese Zuweisung wird typischerweise verwendet, um eine Zählvariable zu initialisieren. Als nächstes wird die Laufbedingung geprüft, und, falls diese zu Wahr ausgewertet wird, der Schleifenkörper ausgeführt. Die zweite Zuweisung wird immer nach dem Durchlaufen des Schleifenkörpers ausgeführt,

und wird üblicherweise zum Erhöhen einer Zählvariable verwendet. Daraufhin wird wieder die Laufbedingung geprüft, und die Schleife wird je nach ermitteltem Wahrheitswert erneut ausgeführt.

```
1 for(i=0; i<20; i=i+1) {
2     print(i, " zum Quadrat ist " + i*i);
3 }
```

3.10. Kommentare

Mit Hilfe von Kommentaren kann der Quelltext dokumentiert werden. Femtoscript unterstützt Zeilenkommentare, die mit zwei Schrägstrichen eingeleitet werden und den gesamten folgenden Rest der Zeile als Kommentar kennzeichnen. Wenn der Interpreter auf einen Kommentar stößt, wird dieser übersprungen, als wenn er aus Whitespace-Zeichen bestünde. Kommentare können prinzipiell an jeder Stelle im Quellcode eingefügt werden, in Stringliteralen werden die beiden Schrägstriche allerdings als Teil des Strings aufgefasst und leiten keinen Kommentar ein.

```
1  // Diese Funktion berechnet das Maximum von a und b
2  function max(a,b) {
3      if(a>b) {
4          return a;
5      } else {
6          return b;
7      }
8  }
9
10 print(max(10,12)); // Ergebnis: 12
```

3.11. Eingebaute Funktionen

Femtoscript verfügt über eine große Zahl eingebauter Funktionen, die hier näher beschrieben sind. Dabei wird jede Funktion zunächst in einer Form vorgestellt, die den Funktionsbeschreibungen aus UML ähnelt und Aufschluss über den Typ und die Zahl der Parameter sowie den Typ des Rückgabewertes gibt.

Da Femtoscript keine Schlüsselwörter für die verschiedenen Typen besitzt, werden die Bezeichnungen „int“, „float“, „string“ und „void“ verwendet, um die Datentypen von Femtoscript in den Funktionsbeschreibungen anzugeben. Zusätzlich zu den tatsächlichen Typen wird „number“ benutzt, um zu kennzeichnen, dass an der entsprechenden Stelle sowohl ein Integer- als auch ein Fließkommawert möglich ist. Ähnlich dazu bedeutet „mixed“, dass jeder Typ außer Void erlaubt ist.

3.11.1. Mathematische Funktionen

abs (number x) : number

Berechnet den Absolutwert von x.

Der Rückgabetyp entspricht dem Typ von x.

cos (number r) : float

Berechnet den Kosinus von r, wobei r als Winkel im Bogenmaß interpretiert wird.

acos (number x) : float

Berechnet den Arkuskosinus von x. Das Ergebnis ist ein Winkel im Bogenmaß aus dem Intervall $[0, \pi]$.

x muss im Intervall $[-1, 1]$ liegen, sonst wird ein Illegal Argument Error ausgelöst.

sin (number r) : float

Berechnet den Sinus von r, wobei r als Winkel im Bogenmaß interpretiert wird.

asin (number x) : float

Berechnet den Arkussinus von x. Das Ergebnis ist ein Winkel im Bogenmaß aus dem Intervall $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

x muss im Intervall $[-1, 1]$ liegen, sonst wird ein Illegal Argument Error ausgelöst.

tan (number r) : float

Berechnet den Tangens von r, wobei r als Winkel im Bogenmaß interpretiert wird.

atan (number x) : float

Berechnet den Arkustangens von x. Das Ergebnis ist ein Winkel im Bogenmaß aus dem Intervall $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

atan2 (number y, number x) : float

Berechnet den Winkel vom Koordinatenursprung zum Punkt (x,y). Das Ergebnis ist ein Winkel im Bogenmaß aus dem Intervall $[-\pi, \pi]$.

Wenn sowohl x als auch y Null sind, wird ein Illegal Argument Error ausgelöst.

exp (number x) : float

Berechnet die Exponentialfunktion. Das Ergebnis ist e^x.

log (number x) : float

Berechnet den natürlichen Logarithmus $\ln(x)$.

Falls x negativ ist, wird ein Illegal Argument Error ausgelöst.

pow (number basis, number exponent) : float

Berechnet $basis^{exponent}$.

In zwei Fällen ist die Funktion nicht definiert und löst einen Illegal Argument Error aus:

- Falls *basis* negativ ist und *exponent* keine ganze Zahl ist, und
- falls *basis* Null ist und *exponent* negativ.

sqrt (number x) : float

Berechnet die Quadratwurzel von x.

Wenn x negativ ist, wird ein Illegal Argument Error ausgelöst.

3.11.2. Zufallsfunktionen

rand () : float

Gibt eine Pseudozufallszahl aus dem Intervall $[0, 1[$ zurück.

rndch (mixed p1 [, mixed p2 [, ...]]) : mixed

Gibt pseudozufällig einen der übergebenen Parameter zurück. Es können bis zu 255 Parameter übergeben werden.

Diese Funktion benutzt intern den gleichen Zufallsgenerator wie *rand()*.

srand (int seed) : void

Setzt den internen Zustand (Seed) des Zufallsgenerators, der für *rand()* und *rndch()* benutzt wird.

Für die meisten Anwendungen ist es hilfreich, den Zufallsgenerator am Anfang mit srand(time()) zu initialisieren.

3.11.3. Konvertierungsfunktionen

str (mixed x) : string

Konvertiert x in einen String.

Fließkommazahlen werden je nach Zahlbereich in Kommaschreibweise oder wissenschaftlicher Notation ausgegeben. Falls x bereits ein String ist, wird x zurückgegeben.

asc (string s) : int

Ermittelt den Bytewert des ersten Zeichens im String s.

Falls s leer ist, wird -1 zurückgegeben.

chr (int a) : string

Erzeugt einen String, der aus einem Zeichen mit dem Bytewert a besteht.

int (mixed x) : int

Wandelt x in eine Integerzahl um.

Falls x ein String ist, wird der Anfang des Strings als Integerzahl interpretiert. Falls keine gültige Integerzahl gefunden wird, wird Null zurückgegeben. Weitere Zeichen hinter einer Integerzahl werden ignoriert. Wenn die Zahl im String zu groß ist, um als Integerwert dargestellt zu werden, ist das Ergebnis undefiniert.

Falls x eine Fließkommazahl ist, wird der Nachkommateil abgeschnitten. Falls x zu groß ist, um durch eine Integerwert dargestellt zu werden, ist das Ergebnis undefiniert.

Falls x bereits den Typ int hat, wird x zurückgegeben.

round (number x) : int

Rundet x auf die nächste ganze Zahl.

Falls x zu groß ist, um als Integerwert dargestellt zu werden, ist das Ergebnis undefiniert.

float (mixed x) : float

Wandelt x in eine Fließkommazahl um.

Falls x ein String ist, wird der Anfang des Strings als Fließkommazahl interpretiert. Falls keine gültige Fließkommazahl gefunden wird, wird Null zurückgegeben. Weitere Zeichen hinter einer Fließkommazahl werden ignoriert.

Falls x bereits eine Fließkommazahl ist, wird x zurückgegeben.

3.11.4. Stringfunktionen

instr (string haystack, string needle) : int

Sucht den String *needle* im String *haystack*.

Falls *needle* nicht in *haystack* enthalten ist, wird -1 zurückgegeben, ansonsten wird der Index des ersten Zeichens des ersten Vorkommens von *needle* in *haystack* zurückgegeben. Dieser Index ist Null-basiert.

len (string s) : int

Gibt die Anzahl der Zeichen im String s zurück.

left (string s, int n) : string

Gibt einen String zurück, der aus den ersten n Zeichen von s besteht.

Falls s weniger als n Zeichen lang ist, wird s zurückgegeben. Falls n kleiner oder gleich Null ist, wird ein leerer String zurückgegeben.

mid (string s, int start, int n) : string

Gibt einen Teilstring von s zurück, der bei Zeichen *start* beginnt und n Zeichen lang ist.

Falls n kleiner oder gleich Null ist, wird ein leerer String zurückgegeben.

Falls der angefragte Substring über das Ende von s hinausgehen würde, wird nur der Substring von Zeichen *start* bis zum Ende von s zurückgegeben.

Falls *start* kleiner als Null oder größer als der Index des letzten Zeichen im String ist, wird ein Illegal Argument Error ausgelöst.

right (string s, int n) : string

Gibt einen String zurück, der aus den letzten n Zeichen von s besteht.

Falls s weniger als n Zeichen lang ist wird s zurückgegeben. Falls n kleiner oder gleich Null ist wird ein leerer String zurückgegeben.

lower (string s) : string

Erzeugt einen neuen String aus s, in dem alle Großbuchstaben durch Kleinbuchstaben ersetzt sind.

upper (string s) : string

Erzeugt einen neuen String aus s, in dem alle Kleinbuchstaben durch Großbuchstaben ersetzt sind.

ltrim (string s) : string

Erzeugt einen neuen String aus *s*, in dem alle Leerzeichen vom Anfang entfernt sind.

trim (string s) : string

Erzeugt einen neuen String aus *s*, in dem alle Leerzeichen vom Anfang und Ende entfernt sind.

3.11.5. Ein/Ausgabefunktionen

print (mixed p1 [, mixed p2 [, ...]]) : void

Gibt alle Parameter nacheinander auf dem Bildschirm aus.

Bei der Ausgabe von Strings wird das Newline-Steuerzeichen interpretiert, so dass print("Hallo \n Welt"); das Wort „Welt“ in eine neue Zeile schreibt.

Die Bildschirmausgabe beginnt an der Position des Print-Cursors. Wenn die Ausgabe über das Ende des Bildschirmspeichers hinausgeht, wird der gesamte Bildschirminhalt nach oben gescrollt, um Platz für die neue Ausgabe zu schaffen.

println (mixed p1 [, mixed p2 [, ...]]) : void

Gibt alle Parameter nacheinander auf dem Bildschirm aus und springt in eine neue Zeile.

Diese Funktion verhält sich genau wie *print()*, setzt aber den Cursor nach der Ausgabe aller Parameter an den Anfang der nächsten Zeile.

locate (int x, int y) : void

Setzt den Print-Cursor an die angegebene Bildschirmposition.

Falls die angegebenen Koordinaten außerhalb des Bildschirms liegen, wird der Cursor in die untere rechte Ecke des Bildschirms gesetzt.

gets (int x, int y) : string

Gibt das Zeichen an der Bildschirmposition (x,y) zurück.

Falls die angegebenen Koordinaten außerhalb des Bildschirmbereichs liegen, wird ein leerer String zurückgegeben.

puts (int x, int y, string s) : void

Kopiert den String s ab Bildschirmposition (x,y) auf den Bildschirm.

Im Gegensatz zur Funktion *print()* wird dabei nur ein Stringparameter akzeptiert, und das Newline-Steuerzeichen wird nicht interpretiert, sondern als Sonderzeichen ausgegeben. Ein weiterer Unterschied ist, dass *puts()* nicht zu einem Scrollen des Bildschirminhalts führt. Falls der auszugebende String über das Ende des Bildschirmspeichers hinausgehen würde, wird das Ende des Strings nicht dargestellt.

cls () : void

Löscht den Bildschirminhalt und setzt den Print-Cursor zurück in die obere linke Ecke des Bildschirms.

input () : string

Fragt eine Benutzereingabe ab und gibt diese als String zurück.

Beim Aufruf dieser Funktion erhält der Benutzer die Möglichkeit, eine Zeichenkette über die Tastatur einzugeben. Die Eingabe wird auf dem Bildschirm angezeigt und kann mit Hilfe der Rücklöschtaste bearbeitet werden. Durch Drücken der Eingabetaste wird die Bearbeitung abgeschlossen, was die Rückgabe der eingetippten Zeichenkette aus der Funktion veranlasst.

get () : string

Liest ein einzelnes Zeichen von der Tastatur.

Diese Funktion prüft, ob der Benutzer ein Zeichen eingegeben hat, und gibt dieses als String zurück. Falls kein Zeichen eingegeben wurde, wird ein leerer String zurückgegeben. Im Gegensatz zur *input()*-Funktion gibt *get()* immer sofort das Ergebnis zurück und wartet nicht auf eine Eingabe.

out (int adresse, int wert) : void

Schreibt ein Byte mit dem Wert *wert* an die Speicherstelle *adresse*.

Mit dieser Funktion lassen sich (in der ATmega-Implementierung) beliebige Speicherstellen manipulieren, um z.B. auf die I/O-Funktionen des ATmega zuzugreifen.

Bei unvorsichtigem Umgang mit dieser Funktion ist es allerdings möglich, das Gerät vollständig zum Absturz zu bringen, so dass es durch Aus- und wieder Einschalten zurückgesetzt werden muss. Der Verlust des im EEPROM gespeicherten Programms ist ebenfalls möglich. Inwieweit eine dauerhafte Beschädigung der Hardware möglich ist, hängt vom konkreten Aufbau des

ATmega-Boards ab: Es könnte beispielsweise möglich sein, einen Kurzschluss zu verursachen, indem zwei miteinander verbundene I/O-Leitungen auf Ausgang geschaltet und auf unterschiedliche Ausgabewerte eingestellt werden.

in (int adresse) : int

Liest den Wert des Bytes aus der Speicherstelle *adresse*.

Mit dieser Funktion lassen sich (in der ATmega-Implementierung) beliebige Speicherstellen auslesen, um z.B. auf die I/O-Funktionen des ATmega zuzugreifen.

Diese Funktion greift zwar nur lesend auf den Speicher zu, allerdings ändern manche I/O-Register des ATmega schon durch lesenden Zugriff ihren Zustand. Insbesondere die Datenregister der seriellen Schnittstellen sind davon betroffen. Das Auslesen der entsprechenden Register kann daher z.B. zur fehlerhaften Interpretation von Tastatureingaben führen.

3.11.6. Zeitfunktionen

time () : int

Gibt einen Zeitstempel in Millisekunden zurück.

Die Startzeit ist nicht festgelegt, und die tatsächliche Granularität des Zeitstempels kann gröber sein. In der ATmega-Implementierung ist die angegebene Zeit die Zeit seit dem Initialisieren der Videoausgabe und hat als zeitliche Auflösung die vertikale Bildwiederholrate.

4. Schnittstellen des Interpreters

In diesem Abschnitt wird erläutert, wie der Interpreter aus Entwicklersicht verwendet werden kann. Dazu werden die Schnittstellen von Femto beschrieben, die wichtig sind, um den Interpreter in eine neue Programmierumgebung oder ein anderes Projekt zu integrieren, oder ihn auf eine neue Plattform zu portieren.

4.1. Starten und Beenden

Der Interpreter wird durch Aufrufen der Funktion *interpretProgram()* ausgeführt, die in der Header-Datei *interpreter.h* im Quelltextverzeichnis des Interpreters definiert ist. Die Funktion ist parameterlos, da der zu interpretierende Programmtext über einen anderen Mechanismus an den Interpreter übergeben wird, der im nächsten Unterabschnitt genauer beschrieben wird.

Um die Ausführung des Interpreters zu unterbrechen, kann aus einer Interruptroutine oder einem anderen Thread heraus die Funktion *interruptInterpreter()* aufgerufen

werden. Diese setzt ein Flag, das den Interpreter dazu veranlasst die Programmausführung zu beenden. Auf dem Einchipcomputer wird diese Funktion aufgerufen, wenn der Benutzer die Tastenkombination Strg+c drückt.

interpretProgram() gibt ein Byte zurück, das anzeigt, aus welchem Grund die Ausführung des interpretierten Programms beendet wurde. Die Codes für die möglichen Ursachen sind ebenfalls in der Datei *interpreter.h* im *enum Returncode* beschrieben. Die anderen Rückgabecodes zeigen verschiedene Fehler an, die zu einem frühzeitigen Abbruch des interpretierten Programms geführt haben. Diese Informationen können verwendet werden, um dem Benutzer eine passende Fehlermeldung anzuzeigen.

4.2. Plattformmodul

Wie bereits in der Übersicht beschrieben wurde, muss der Interpreter gegen mehrere plattformspezifische Headerdateien kompiliert werden, die Unterschiede verschiedener Plattformen vom Interpreter abkapseln. Die Headerdateien für die richtige Plattform müssen beim Kompilieren des Interpreters im Includepfad <environment/> erreichbar sein. Sie befassen sich mit

- Quellcodeverwaltung (*codestorage.h*),
- Bildschirmausgabe (*screen.h*),
- Tastatureingabe (*keyboard.h*),
- Zeitabfrage (*time.h*) und
- Makros für C-Stringliterale (*literalStringfunctions.h*).

Die Quelldateien, die diese Funktionalität bereitstellen, werden hier als Plattformmodul bezeichnet.

Im Projektverzeichnis von Femto (im Verzeichnis *src/platforms/template/*) befindet sich eine Vorlage, die nach Bedarf für die gewünschte Zielplattform angepasst werden kann. Diese Beispielimplementierung definiert den zu interpretierenden Quelltext direkt als Stringliteral, verwendet ein statisch definiertes Byte-Array zur Simulation der Bildschirmausgabe und benutzt Standard-C-Funktionen für die Zeitabfrage und für Stringfunktionen. Das Abfragen der Tastatur wird nicht unterstützt; die entsprechenden Funktionen verhalten sich immer so, als läge keine Eingabe vor.

4.2.1. Quellcodeverwaltung

Femto wurde für kleine Systemen wie den Einchipcomputer entwickelt, die über wenig RAM verfügen. Dies war der Grund für mehrere Entwurfsentscheidungen, die den Speicherbedarf des Interpreters verringern sollten. Dazu gehört, dass große Quelltexte nicht vollständig in den Hauptspeicher passen müssen, sondern in Abschnitten (z.B. zeilenweise) von einem externen Speicher nachgeladen werden können.

Dafür könnten verschiedene Mechanismen verwendet werden, etwa das Lesen aus einer Datei auf einer SD-Speicherkarte oder aus einem externen EEPROM-Baustein. Da

Tabelle 4: Rückgabecodes des Interpreters und deren Bedeutung

Rückgabecode	Bedeutung
ERROR_NONE	Das Programm wurde normal beendet.
ERROR_BREAK	Der Interpreter wurde durch Aufruf von *interruptInterpreter()* unterbochen.
ERROR_IDENTIFIER_TOO_LONG	Das Programm enthält einen Bezeichner, der länger als 255 Zeichen ist.
ERROR_ILLEGAL_ARGUMENT	Eine eingebaute Funktion wurde mit unpassenden Werten aufgerufen (z.B. acos(5)).
ERROR_DIVISION_BY_ZERO	Im Programm wurde versucht, eine Division durch Null durchzuführen.
ERROR_MISMATCHED_PARENTHESIS	Ein Ausdruck enthält eine schließende Klammer, die keiner öffnenden Klammer zugeordnet werden kann.
ERROR_NUM_PARAMETERS	Eine Funktion wurde mit der falschen Zahl an Parametern aufgerufen.
ERROR_OUT_OF_MEMORY	Das Programm benötigt zur Ausführung mehr Stack- oder Stringspeicher, als dem Interpreter zur Verfügung steht.
ERROR_STRING_REFERENCE_OVERFLOW	Der gleiche String wird mehr als 255 mal referenziert. Dies ist eine Einschränkung der verwendeten String-Implementierung, die im Abschnitt 5.4.2 genauer erläutert wird.
ERROR_STRING_TOO_LONG	Es wurde versucht, eine Zeichenkette mit mehr als 255 Zeichen zu erzeugen.
ERROR_SYNTAX_ERROR	Der Programmtext enthält ein Zeichen, das an dieser Stelle nicht zulässig ist.
ERROR_TYPE_MISMATCH	Bei einem Operator oder einer eingebauten Funktion wurde ein unpassender Datentyp verwendet (z.B. Division mit Strings). Der Typ Void löst bei jeder Verwendung diesen Fehler aus.
ERROR_UNEXPECTED_END_OF_PROGRAM	Der Interpreter hat das Ende des Programmquelltextes erreicht, aber es wurden noch weitere Zeichen erwartet.
ERROR_UNKNOWN_FUNCTION	Es wurde versucht, eine Funktion aufzurufen die nicht definiert ist.
ERROR_UNKNOWN_VAR	Es wurde versucht, auf eine Variable zuzugreifen die nicht definiert ist.
ERROR_INTERNAL	Der Interpreter enthält einen Programmierfehler.

die verwendete Umsetzung von der Plattform abhängt, wird der Quelltext durch das Plattformmodul bereitgestellt.

Der Interpreter ruft beim Starten die Funktion *loadFirstLine()* aus dem Plattformmodul auf, die den ersten Abschnitt des Programmtextes in einen Puffer läd und einen Zeiger auf das erste Zeichen zurückgibt. Wie dieser Puffer angelegt und verwaltet wird bleibt dem Plattformmodul überlassen. Der Puffer ist wie ein C-String durch ein Nullbyte abgeschlossen, das jedoch nur das Ende des aktuellen Abschnitts signalisiert. Um den nächsten Abschnitt in den Puffer zu laden, wird die Funktion *loadNextLine()* verwendet. Diese gibt einen Zeiger auf das erste Zeichen im nächsten Block zurück, oder NULL, falls das Ende des Quelltextes erreicht ist.

Ein Abschnitt darf nicht an jeder beliebigen Stelle im Quelltext enden, sondern nur dort, wo Whitespacezeichen stehen oder eingefügt werden könnten, ohne die Bedeutung des Programmcodes zu ändern. Bezeichner, Schlüsselwörter, Literale und Operatorsymbole aus mehreren Zeichen dürfen also nicht „abgeschnitten“ werden. Eine Ausnahme bilden Stringliterale, die an jeder Stelle aufgeteilt werden dürfen. Da ein Abschnitt an jeder Stelle enden darf, an der ein Whitespacezeichen im Quelltext steht, bietet es sich an, den Quelltext immer an bestehenden Leerzeichen oder am Zeilenende aufzuspalten. Die Länge einer Programmzeile in Femto ist allerdings nicht beschränkt, man sollte daher eine maximale Zeilenlänge für die entsprechende Plattform festlegen, um sicherzustellen, dass immer genug Speicher vorhanden ist, um die nächste Zeile zu laden.

Der Interpreter muss oft an eine andere Stelle im Quelltext springen, z.B. um eine Schleife zu wiederholen. Mit den bisher beschriebenen Funktionen lässt sich der Quelltext aber nur linear lesen. Um Sprünge zu erlauben, muss das Plattformmodul zwei weitere Funktionen zur Verfügung stellen, *getParseposition()* und *loadParseposition()*, mit deren Hilfe sich der Interpreter eine Position im Quelltext merken und später wieder dorthin zurückspringen kann.

getParseposition() erzeugt eine Referenz auf die aktuell betrachtete Quelltextstelle und gibt diese an den Interpreter zurück. *loadParseposition()* nimmt eine solche Referenz entgegen, läd einen Abschnitt, der die gewünschte Stelle enthält, und gibt einen Zeiger auf diese Stelle zurück. Da der Inhalt einer solchen Referenz davon abhängt, wie die Quellcodeverwaltung umgesetzt ist, wird zusätzlich der Datentyp *Parseposition* für diese Referenzen definiert.

Die Funktionen der Quellcodeverwaltung werden in der Datei *codestorage.h* deklariert.

4.2.2. Bildschirmausgabe

Die Schnittstelle zur Bildschirmausgabe wurde entworfen, um die Verwendung von verschiedenartigen Anzeigen zu ermöglichen. Es wird allerdings davon ausgegangen, dass das Anzeigegerät eine feste Anzahl von Zeichen pro Zeile und eine feste Anzahl von Zeilen besitzt. Anzeigen mit Proportionalschrift werden also nicht voll unterstützt. In der Datei *screen.h* des Plattformmoduls müssen Höhe und Breite des Bildschirms definiert werden. *screenWidth* gibt die Anzahl der Zeichen pro Zeile an, *screenHeight* die Anzahl der Zeilen. Diese Symbole können als Variablen deklariert oder mit Hilfe von #define-Anweisungen konstant angegeben werden.

In der gleichen Datei muss die Integervariable *screenCursor* deklariert werden. Diese Variable enthält den Index des nächsten Zeichens auf dem Bildschirm, das durch den Aufruf einer print-Funktion überschrieben wird. Der Interpreter kann den Wert dieser Variable ändern, um die Position der nächsten Ausgabe zu beeinflussen.

Zusätzlich müssen in der Datei *screen.h* sechs Funktionen deklariert werden.

screenClear() löscht den Bildschirminhalt, indem der Bildschirm mit Leerzeichen gefüllt wird. *screenPut()* kopiert den Inhalt eines übergebenen Puffers ab einer vorgegebenen Position in den Bildschirmspeicher, und *screenGet()* liest das Zeichen an der vorgegebenen Position aus dem Bildschirmspeicher.

screenPrintInt() und *screenPrintFloat()* schreiben ab der Position, die durch *screenCursor* festgelegt ist, einen Integerwert bzw. einen Fließkommawert auf den Bildschirm. Falls hinter der Cursorposition nicht mehr genug Platz ist, um den Wert auf dem Bildschirm auszugeben, wird der Bildschirminhalt zuerst nach oben gescrollt. Nach der Ausgabe wird *screenCursor* hinter das letzte ausgegebene Zeichen gesetzt. *screenPrintStr()* schreibt einen String auf den Bildschirm, wobei ebenfalls *screenCursor* zur Positionierung verwendet wird und bei Bedarf gescrollt wird.

Es fällt auf, das mit *screenPrintStr()* und *screenPut()* zwei verschiedene Funktionen zur Ausgabe von Strings implementiert werden müssen. Ein Unterschied zwischen diesen Funktionen ist, dass *screenPut()* den übergebenen String ohne aufwändige Verarbeitung an eine vorgegebene Stelle auf den Bildschirm kopiert. Falls der String zu lang ist und nicht vollständig in den Bildschirmspeicher passt, wird kein Scrolling durchgeführt. Statt dessen erscheint nur der Anfang des Strings auf dem Bildschirm. Abgesehen davon interpretiert *screenPrintStr()* das Newline-Steuerzeichen, indem der Cursor in eine neue Bildschirmzeile gesetzt wird. *screenPut()* kopiert dieses Zeichen stattdessen wie jedes andere, so dass es als Sonderzeichen erscheinen kann.

Beide Funktionen haben allerdings gemeinsam, dass die übergebenen Strings ISO 8859-15 als Zeichensatz benutzen. Die Bildschirmausgabe sollte also die in diesem Standard definierten Zeichen für entsprechende Bytewerte ausgeben. Bytewerte, deren Bedeutung vom Standard nicht vorgegeben wird, können als beliebige Sonderzeichen dargestellt werden.

Die Beispielimplementierung in der oben besprochenen Vorlage reserviert einen Speicherbereich im RAM als Bildschirmspeicher und setzt darauf die Funktionalität aller sechs Funktionen um. Es sind also nur kleine Anpassungen notwendig, um diese Implementierung für eine Plattform zu verwenden, die einen Bildschirmspeicher im Arbeitsspeicher verwendet.

4.2.3. Tastatureingabe

In der Datei *keyboard.h* muss nur eine Funktion bereitgestellt werden: *keybNextChar()*. Falls eine Taste auf der Tastatur gedrückt wurde, soll diese Funktion das entsprechende Zeichen zurückgeben. Falls keine Taste gedrückt wurde gibt die Funktion -1 zurück.

Druckbare Zeichen sollten dabei nach ISO 8859-15 kodiert werden. Für Steuertasten wird die folgende Konvention verwendet:

Taste	Code
Eingabetaste	10 (ASCII-Linefeed)
Rücklöschtaste	8 (ASCII-Backspace)

Diese Zuordnung wird von der eingebauten *input()*-Funktion vorausgesetzt, darüber hinaus findet im Interpreter keine Auswertung von Steuerzeichen statt. Andere Steuertasten können also beliebig den verbleibenden unbenutzten Bytewerten der ISO 8859-Kodierung zugeordnet werden.

4.2.4. Zeitabfrage

Im C-Standard sind zwar Funktionen zur Bestimmung der aktuellen Zeit vorgesehen, diese sind aber in der C-Bibliothek avr-libc, die bei der Entwicklung des Interpreters für den ATmega verwendet wurde, nicht umgesetzt. Daher wurde diese Funktionalität ebenfalls ins Plattformmodul übertragen.

Dazu muss in der Headerdatei *time.h* die Funktion *timeGetMilliseconds()* zur Verfügung gestellt werden, die einen 32-Bit Zeitstempel in Millisekunden zurückgibt. Es muss keine sinnvolle absolute Zeitangabe gemacht werden (z.B. Millisekunden seit Mitternacht). Stattdessen reicht es, wenn die Differenz von zwei abgefragten Zeitstempeln der Zeit zwischen den Aufrufen von *timeGetMilliseconds()* entspricht.

Die tatsächliche Auflösung des Zeitgebers muss nicht Millisekundengenau sein, auch wenn eine möglichst feine Auflösung wünschenswert ist. Der ermittelte Zeitwert wird nicht vom Interpreter ausgewertet, sondern von der Femtoscript-Funktion *time()* direkt an das interpretierte Programm weitergegeben.

In der Template-Implementierung werden die Standard C-Funktionen zur Bestimmung eines Zeitstempels verwendet.

4.2.5. Makros für C-Stringliterale

Viele C-Compiler reservieren für im Quelltext verwendete Stringliterale statisch Arbeitsspeicher, obwohl Stringliterale konstant sind und auf Plattformen wie dem ATmega im Flash-Speicher untergebracht werden könnten. Dieses Verhalten dient der Kompatibilität zu den C-Standards, ist aber hier ungünstig, weil dabei eine große Menge Arbeitsspeicher dauerhaft belegt wird.

Die avr-libc enthält Makros, mit deren Hilfe Stringliterale im Flash-Speicher angelegt werden können. Die Adressen der Strings beziehen sich dann auf den Flash-Speicher anstatt den Arbeitsspeicher. Da diese beiden Arten von Adressen sich aber aus Sicht des Compilers nicht unterscheiden, liefert die avr-libc zusätzliche Stringfunktionen, die als Parameter Zeiger auf den Programmspeicher erwarten.

Die direkte Verwendung dieser avr-libc-spezifischen Makros und Funktionen im Interpreter widerspricht der angestrebten Plattformunabhängigkeit. Daher wurde die Headerdatei *literalStringfunctions.h* für das Plattformmodul eingeführt, in der die zu verwendenden Stringfunktionen festgelegt werden können. Auf einer Plattform wie dem PC sollten hier die Standard C-Stringfunktionen verwendet werden.

In Femto werden Stringliterale nur im Zusammenhang mit Vergleichen und als Formatstrings benutzt. Daher werden in der Datei *literalStringfunctions.h* die Funktionsartigen Makros *literalStringStrncmp()* und *literalStringSnprintf()* definiert, die die Funktionalität der Funktionen *strncmp()* und *snprintf()* aus der Standard-Headerdatei <string.h> bereitstellen sollen und auch die gleichen Parameter wie diese Funktionen annehmen.

literalStringStrncmp() erhält immer ein Stringliteral als zweiten Parameter. Dieses Literal wird in der ATmega-Version der *literalStringfunctions.h* so modifiziert, dass es als String im Flash-Speicher angelegt wird. Ähnlich dazu wird auch bei *literalStringSnprintf()* immer ein Stringliteral als Formatstring erwartet.

5. Funktionsweise des Interpreters

5.1. Speichernutzung

Da der Arbeitsspeicher des im Einchipcomupter eingesetzten ATmega644 vergleichsweise knapp ist, musste beim Entwickeln von Femto besonders darauf geachtet werden, mit dieser Ressource sparsam umzugehen. Daher ist es wichtig zu verstehen, wie der Interpreter den zur Verfügung stehenden Speicherplatz nutzt. Man kann den verwendeten Speicher in vier verschiedenen Kategorien unterteilen.

Als erstes belegt der Interpreter wie jedes Programm Speicher für seinen ausführbaren Maschinencode. Auf dem ATmega wird dieser Programmcode im Flash-Speicher abgelegt. Da auf dem ATmega644 eine große Menge an Flash-Speicher verfügbar ist, wurde bei der Entwicklung nicht darauf geachtet, den Speicherbedarf in dieser Kategorie zu minimieren.

C-Programme legen lokale Variablen und Rücksprungadressen in einer Stack-Struktur ab. Beim Betreten einer Funktion wächst dieser Stack, beim Verlassen wird wieder Speicher freigegeben. Dieser Stack wird im Folgenden C-Stack genannt und ist die zweite Speicherkategorie, die hier betrachtet wird.

Der genaue Aufbau des C-Stack ist hier nicht weiter interessant, das Verhalten bei verschachtelten Funktionsaufrufen dagegen schon, denn bei steigender Aufruftiefe wächst der C-Stack immer weiter an. Dabei ergeben sich zwei Probleme. Erstens ist es aufwändig sicherzustellen, dass der Stackspeicher nicht überläuft. Das Ergebnis eines solchen Überlaufs hängt von der Plattform ab, es ist aber davon auszugehen, dass ein sauberes Beenden des Interpreters nach einem Überlauf nicht mehr möglich wäre. Zweitens wächst der Stack oft sehr schnell an, so dass rekursive Funktionsaufrufe viel Arbeitsspeicher benötigen. Aus diesem Grund werden rekursive Funktionsaufrufe in Femto vollständig vermieden. Die Aufruftiefe der Funktionen wird ebenfalls flach gehalten. Die dazu verwendete Technik wird im Abschnitt 5.3.2 näher beschrieben.

Als nächste Speicherkategorie folgen globale C-Variablen. Diese werden durch den C-Compiler statisch reserviert und belegen dadurch dauerhaft Arbeitsspeicher. Femto verwendet nur wenige globale Variablen.

Abgesehen davon verwendet Femto auch dynamisch reservierten Speicher, der mit *malloc()* und *free()* reserviert und freigegeben wird. Das ist die vierte Kategorie. Beim

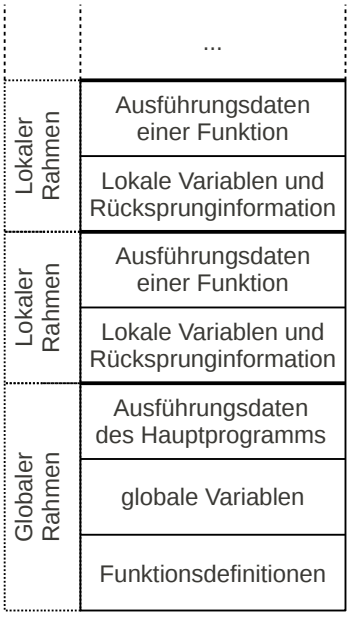

Abbildung 3: Übersicht über den Aufbau des Interpreterstacks während der Programmausführung

Starten des Interpreters wird zunächst ein Speicherbereich für den Interpreterstack reserviert. Der Interpreterstack wird im folgenden Abschnitt genauer beschrieben. In der weiteren Ausführung wird je nach interpretiertem Programm auch dynamisch Speicherplatz für Zeichenkettenvariablen (Abschnitt 5.4.2) reserviert.

5.2. Der Interpreterstack

Der Interpreterstack ist eine Datenstruktur, in der verschiedene Teile des Interpreters Informationen zu Variablen, Funktionen und dem Ausführungszustand des interpretierten Programms ablegen. Der Stack wächst von niedrigen zu hohen Adressen. Zum vereinfachten Zugriff auf diese Datenstruktur werden in der Datei *programstack.h* mehrere Makros definiert, die Daten von verschiedenen Typen auf dem Stack ablegen und wieder von Stack lesen können.

Wie bereits oben erwähnt, wird für den Interpreterstack beim Start des Interpreters ein Speicherbereich mit fester Größe reserviert. Die Größe kann vor dem Kompilieren des Interpreters in der Datei *config.h* eingestellt werden und beträgt normalerweise 512 Bytes. Beim Ablegen von Daten auf dem Interpreterstack wird geprüft, ob noch genug Speicherplatz vorhanden ist; falls nicht, wird der Interpreter mit einem Fehlercode beendet.

Abbildung 3 gibt eine Übersicht über den Aufbau des Interpreterstacks. Beim Start des Interpreters werden zuerst die Funktionsdefinitionen eingelesen. Dabei wird auf dem

Interpreterstack für jede Funktion der Bezeichner sowie ein Verweis auf die Position im Quelltext abgelegt. Diese Tabelle wird bei Funktionsaufrufen verwendet, um die Position der gewünschten Funktion zu finden.

Auf die Tabelle der Funktionen folgt die der globalen Variablen. Hier werden die Bezeichner und Werte der globalen Variablen abgelegt. Dahinter folgen Informationen, die während der Ausführung des Hauptprogramms abgelegt werden. Das sind Hinweise auf die nächsten Sprachkonstrukte, die der Interpreter aufgrund des bereits abgearbeiteten Programmcodes erwartet (siehe Abschnitt 5.3.2), falls im Hauptprogramm gerade ein Ausdruck ausgewertet wird befindet sich in diesem Bereich auch der aktuelle Zustand der Ausdrucksauswertung (siehe Abschnitt 5.5).

Der gesamte untere Teil des Interpreterstacks, bestehend aus den Tabellen der Funktionen und globalen Variablen zusammen mit den Ausführungsdaten des Hauptprogramms, bildet den globalen Rahmen des Interpreterstacks. Oberhalb davon können sich lokale Rahmen befinden, die die lokalen Variablen, Ausführungsdaten und Rücksprunginformationen von aufgerufenen Funktionen enthalten. Bei jedem Funktionsaufruf wird ein solcher Rahmen oben auf dem Interpreterstack angelegt, bei jedem Rücksprung aus einer Funktion wird der gesamte oberste Rahmen wieder entfernt.

5.3. Ablauf der Ausführung des Interpreters

In Abschnitt 4.1 wurde bereits erklärt, wie der Interpreter aus Sicht des aufrufenden Programms gestartet und beendet wird. Nun wird betrachtet, wie sich der Interpreter beim Aufruf der Funktion *interpretProgram()* initialisiert, wie die Hauptschleife des interpreters aufgebaut ist und wie sichergestellt wird, dass der Interpreter sich im Fehlerfall sauber beendet.

Der Quelltext für Initialisierung, Termination und Hauptschleife befindet sich in der Datei *interpreter.c*.

5.3.1. Start

Beim Initialisieren werden zunächst einige globale Variablen auf ihre Ausgangswerte gesetzt. Abgesehen von mehreren Zeigern, die auf NULL initialisiert werden, wird auch ein Flag für die Unterbrechung des Interpreters zurückgesetzt, die Variable *accumulator*, in der Ergebnisse bei der Auswertung von Ausdrücken zwischengespeichert werden, wird gelöscht, und der Print-Cursor wird in die obere linke Ecke des Bildschirms verschoben. Anschließend wird der Bildschirminhalt gelöscht, so dass ein interpretiertes Programm immer auf einem leeren Bildschirm startet.

Als nächstes wird der Sprungpuffer *exceptionHandler* angelegt, der für die Fehlerbehandlung verwendet wird. Der nächste Abschnitt geht darauf genauer ein. Schließlich wird der Speicherplatz für den Interpreterstack reserviert, und der erste Block des zu interpretierenden Programmtextes wird geladen. Der Zeiger auf den Anfang des Quelltextes wird in der Variable *parsePtr* abgelegt, die immer auf die aktuell vom Interpreter betrachtete Stelle im Programmtext zeigt.

Nun beginnt der Interpreter mit der Auswertung des Programmtextes. Zuerst werden alle Funktionsdefinitionen gelesen. Der Interpreter erwartet dabei immer zuerst das Schlüsselwort **function**, liest dann den darauf folgenden Bezeichner der Funktion und legt ihn auf dem Interpreterstack ab. Anschließend ermittelt der Interpreter die aktuelle Position im Programmtext um später zum Aufrufen der Funktion dorthin springen zu können. Die Position wird ebenfalls auf dem Interpreterstack abgelegt. Abschließend wird der Funktionskörper übersprungen, wobei aus Effizienzgründen nur die geschweiften Klammern beachtet werden, die den Funktionskörper begrenzen. Dieser Vorgang wird solange wiederholt, bis der Interpreter nicht mehr auf das Schlüsselwort **function** trifft. Auf diese Weise entsteht an unterster Stelle auf dem Interpreterstack eine Liste der benutzerdefinierten Funktionen, die später verwendet werden kann, um die Funktionen aufzurufen.

5.3.2. Die Hauptschleife

Nach der Initialisierung werden die Bytewerte der Konstanten *INTERPRET_END* und *INTERPRET_FUNCTION_BODY* auf dem Interpreterstack abgelegt, danach betritt der Interpreter die Hauptschleife und beginnt damit, die globalen Variablendefinitionen und das Hauptprogramm zu interpretieren.

In der Hauptschleife wird bei jedem Durchlauf ein Byte vom Interpreterstack genommen, das die nächste zu erledigende Aufgabe beschreibt. In einer großen switch-Anweisung über den Wert des gelesenen Bytes wird daraufhin in den meisten Fällen eine Funktion aufgerufen, die diese Aufgabe erfüllen soll. Der Name der aufgerufenen Funktion entspricht immer dem Namen der Konstante, die die Aufgabe beschreibt, abgesehen von der unterschiedlichen Schreibkonvention von Konstanten und Funktionen.

Die beiden Bytes, die vor Beginn der Hauptschleife auf den Interpreterstack gelegt wurden, repräsentieren die beiden grundlegenden Aufgaben, die der Interpreter noch erfüllen muss: Interpretiere einen Funktionskörper (das Hauptprogramm) und beende danach den Interpreter. Die scheinbar vertauschte Reihenfolge, mit der diese Aufgaben auf den Interpreterstack gelegt werden (Zuerst *INTERPRET_END*, dann *INTERPRET_FUNCTION_BODY*) rührt daher, dass durch die verwendete Stapelstruktur das zuletzt geschriebene Byte zuerst von der Hauptschleife ausgewertet wird.

Um die gegebene Aufgabe zu erfüllen, können die aufgerufenen Funktionen weitere Aufgaben auf dem Interpreterstack ablegen. Da in der Hauptschleife immer nur das oberste Byte vom Interpreterstack gelesen und dann die gewünschte Funktion aufgerufen wird, können direkt unter diesem Byte Parameterdaten abgelegt werden, die dann von der aufgerufenen Funktion wieder vom Interpreterstack gelesen werden können.

Das Übergeben von Rückgabewerten ist nicht vorgesehen, statt dessen wirken sich die aufgerufenen Funktionen direkt auf den Zustand des Interpreters aus (Programmzeiger, Inhalt der Variablen des interpretierten Programms) oder speichern ihr Ergebnis in der globalen Variable *accumulator*, die den gleichen Datentyp besitzt wie die Variablen in Femtoscript und als Zwischen- und Endergebnis bei der Ausdrucksevaluierung benutzt wird.

Das beschriebene Verfahren erscheint auf den ersten Blick vermutlich unnötig kompli-

ziert und anfällig für Programmierfehler. Es wäre naheliegender, die nötigen Funktionen direkt aufzurufen, anstatt eine Aufforderung auf dem Interpreterstack abzulegen, und den tatsächlichen Aufruf von einer Hauptschleife durchführen zu lassen. Dass Femto dennoch die oben beschriebene Technik verwendet, hängt mit der Speicherknappheit auf dem ATmega zusammen. Der Vorteil liegt darin, dass Rekursion vermieden werden kann.

Die naheliegende Art der Programmierung führt zu Rekursion. Wenn es beispielsweise im Interpreter eine Funktion gibt, um for-Schleifen auszuführen, und im interpretierten Programm zwei verschachtelte for-Schleifen benutzt werden, wird diese Funktion rekursiv aufgerufen. Mehrere Verschachtelungen dieser Art führen zu einem immer größeren Speicherbedarf des C-Stacks. Auf einem modernen PC stellt das normalerweise kein Problem dar, aber auf einer Plattform mit sehr wenig Hauptspeicher ergeben sich mehrere Schwierigkeiten.

Zum einen speichert der bei der Entwicklung des Interpreters verwendete Compiler avr-gcc bei Funktionsaufrufen deutlich mehr Daten auf dem C-Stack als nötig, so dass der auf dem ATmega zur Verfügung stehende Speicher bereits bei geringer Rekursionstiefe nicht mehr ausreicht. Zum anderen gibt es keine einfache Möglichkeit, Speicherknappheit zu erkennen, um einen Überlauf des C-Stacks zu verhindern und den Interpreter sauber zu beenden. Mit der oben beschriebenen Technik tritt dieses Problem nicht auf. Rekursion kann vollständig vermieden werden, so dass der maximal für den C-Stack benötigte Speicherplatz beschränkt ist.

Wie man mit dieser Technik programmiert, soll an einem Beispiel verdeutlicht werden: Um die Bedingung in einem if-Statement zu prüfen muss der Bedingungsausdruck ausgewertet werden, abhängig vom Ergebnis wird dann der folgende Code-Block übersprungen oder ausgewertet. Das könnte man so implementieren:

```
1  void interpretIf() {
2      // Nach dem "if" muss eine oeffnende Klammer folgen
3      expect('(');
4      if(interpretExpression() == true) {
5          // Nach dem Ausdruck muss eine
6          // schliessende Klammer folgen
7          expect(')');
8          interpretCodeblock();
9      } else {
10         // Nach dem Ausdruck muss eine
11         // schliessende Klammer folgen
12         expect(')');
13         skipCodeblock();
14     }
15 }
```

Die Möglichkeit eines folgenden else-Blocks wird hier zur Vereinfachung ignoriert. Der gezeigte Ansatz ist rekursiv, da ein Ausdruck Funktionsaufrufe beinhalten kann, die wieder if-Blöcke enthalten, was zu einem rekursiven Aufruf der Funktion *interpretIf()* führen

würde. Diese Rekursion lässt sich jedoch mit der oben beschriebenen Technik vermeiden. Dafür muss die If-Behandlung allerdings in zwei Funktionen aufgeteilt werden:

```
1  interpretIf() {
2      // Nach dem "if" muss eine oeffnende Klammer folgen
3      expect('(');
4
5      // Fuehre die Funktion "interpretIfFirstBlock" aus
6      // sobald der hier folgende Ausdruck ausgewertet ist.
7      push(INTERPRET_IF_FIRST_BLOCK);
8      push(INTERPRET_EXPRESSION);
9  }
10
11 interpretIfFirstBlock() {
12     // Nach dem Ausdruck muss eine
13     // schliessende Klammer folgen
14     expect(')');
15
16     if(accumulator == true) {
17         push(INTERPRET_CODEBLOCK);
18     } else {
19         push(SKIP_CODEBLOCK);
20     }
21 }
```

Nach dem Aufruf von interpretIf() befinden sich zwei Aufgaben auf dem Stack. Dabei liegt das Auswerten eines Ausdrucks zuoberst und wird damit im nächsten Durchlauf der Hauptschleife ausgeführt. interpretExpression() schiebt selbst wieder Aufrufe auf den Stack, um Operanden auszuwerten oder um Funktionen aufzurufen. Wenn diese Aufgaben vollständig abgearbeitet sind, erscheint *INTERPRET_IF_FIRST_BLOCK* als oberster Stack-Wert, was den Aufruf von interpretIfFirstBlock() veranlasst.

5.3.3. Ende bei fehlerfreier Programmausführung

Wenn die Programmausführung am Ende des Hauptprogramms angekommen ist, verbleibt nur noch die Aufgabe *INTERPRET_END* auf dem Stack, die bereits zu Beginn der Ausführung dort abgelegt wurde. Alternativ dazu legt auch die Funktion, die die return-Anweisung auswertet, *INTERPRET_END* auf dem Stack ab, falls das return-Statement im Hauptprogramm ausgeführt wurde. Sobald die Hauptschleife diesen Wert vom Stack liest, wird der Interpreter beendet.

Der Interpreter beendet sich, indem er zunächst allen reservierten Speicher freigibt. Zum Einen ist das der Speicher für den Interpreterstack, der beim Start angelegt wurde, zum Anderen auch Speicher, der während der Ausführung für Strings reserviert wurde. Abschließend wird der Rückgabecode *ERROR_NONE* aus der Funktion *interpretProgram()* zurückgegeben.

5.3.4. Fehlerbehandlung

Während dem Interpretieren des Programms kann der Interpreter auf unterschiedliche Probleme stoßen, die zum Abbruch des interpretierten Programms führen müssen. Manche dieser Probleme gehen auf Fehler im Quelltext zurück (Syntaxfehler, Falsche Parameterzahl bei Funktionsaufrufen), andere auf Umstände die erst bei der Ausführung ersichtlich werden (Stapelüberlauf, Wertebereichfehler). In allen Fällen ist es wichtig, dass der Interpreter sich sauber, also ohne unerwünschte Seiteneffekte und ohne Speicherlecks beendet und dem aufrufenden Programm einen entsprechenden Fehlercode liefert.

Da der Interpreter in C geschrieben ist, und C im Gegensatz zu C++ keine Exceptions kennt, muss man hier eine alternative Lösung finden. Ein möglicher Ansatz ist, jeder Funktion einen speziellen Rückgabewert für Fehlerfälle zuzuweisen, oder die globale errno-Variable aus der C-Bibliothek zu setzen, wenn ein Fehler auftritt. In den aufrufenden Funktionen müsste dann aber nach jedem Funktionsaufruf geprüft werden, ob ein Fehler aufgetreten ist. Falls ja, müsste die Funktion selbst wieder einen Fehlercode zurückliefern. Dieser Ansatz führt deshalb dazu, dass ein beachtlicher Teil des Quelltextes in jeder Funktion sich nur mit dem Überprüfen von Rückgabewerten befasst. Dadurch wird der Quelltext unübersichtlich und schwer verständlich, und die Effizienz des Programms sinkt durch die ständigen Überprüfungen ebenfalls.

Die C-Standardbibliothek bietet allerdings eine andere Möglichkeit zur Ausnahmebehandlung: Nichtlokales goto ermöglicht es, über Funktionsgrenzen hinaus zu einer anderen Stelle im Programm zu springen. Dazu dienen die Funktionen *setjmp()* und *longjmp()*, die in der Datei *setjmp.h* definiert sind. Mit *setjmp()* kann man den aktuellen Status des Stack- und des Programmzeigers sowie den Zustand aller Register in einem Puffer speichern. Mit *longjmp()* kann man durch Angabe dieses Puffers wieder an die Stelle des Aufrufs von *setjmp()* zurückspringen. Das Programm wird dann so weiter ausgeführt, als sei gerade die *setjmp()*-Funktion aufgerufen worden. Man muss daher beim Aufruf von *setjmp()* den Rückgabewert prüfen, um zu unterscheiden, ob gerade der Rücksprungpuffer angelegt wurde, oder ob ein Sprung erfolgt ist.

Im Interpreter wird ein Rücksprungpuffer in der globalen Variable *exceptionHandler* bereitgestellt, der überall im Interpreter genutzt wird, wo ein kritischer Fehler auftreten kann. Nach dem Rücksprung wird wie bei der normalen Beendigung des Interpreters der gesamte vom Interpreter reservierte Speicher freigegeben. Allerdings gibt die Interpreterfunktion nach dem Aufräumen einen Fehlercode zurück.

Nach einem solchen Abbruch ist es oft interessant, an welcher Stelle im interpretierten Programm der Fehler aufgetreten ist. In der aktuellen Implementierung der Quellcodeverwaltung reicht es dafür aus, die Position von *parsePtr* im Quelltext zu ermitteln, da dieser Zeiger nach der Ausführung nicht zurückgesetzt wird und damit immer noch an die letzte vom Interpreter betrachtete Speicherstelle zeigt. Der ATmega-Programmeditor setzt auf diese Weise den Bearbeitungscursor nach einem Fehler auf die Fehlerstelle.

```
1  union UntypedValue {
2      int32_t i32;
3      float f;
4      struct StringVar str;
5  };
6
7  struct TypedValue {
8      uint8_t type;
9      union UntypedValue val;
10 };
```

Abbildung 4: Auszug aus der Headerdatei typedValue.h

5.3.5. Ende durch Benutzerabbruch

Wie bereits im Abschnitt über die Schnittstellen des Interpreters angesprochen wurde, ist es möglich, die Ausführung des Interpreters zu unterbrechen, indem in einer Interruptroutine oder in einem anderen Thread die Funktion *interruptInterpreter()* aufgerufen wird. Diese Funktion setzt ein Flag im Interpreter, das bei jedem Durchlauf der Hauptschleife (und während der Ausführung der eingebauten input-Funktion) geprüft wird. Falls der Interpreter feststellt, dass das Flag gesetzt ist, wird die Ausführung abgebrochen, indem der Fehler *ERROR_BREAK* ausgelöst wird.

5.4. Datentypen und Variablen

5.4.1. Interne Repräsentation von Werten

Wie bereits im Abschnitt 3.3 beschrieben besitzt ein Femtoscript-Wert einen von vier verschiedenen Datentypen. Im Interpreter wird ein solcher Wert mit Hilfe des C-Typs *struct TypedValue* dargestellt, der sowohl den Typ als auch den eigentlichen Wert eines Femtoscript-Wertes enthält[2]. Sowohl beim Bearbeiten von Femtoscript-Werten als auch beim Ablegen von Werten in Variablen wird das Format dieses *struct*s verwendet. Die verwendeten Konstanten und Datentypen sind in der Headerdatei *typedValue.h* definiert.

Das erste der beiden Elemente in *TypedValue* ist das Byte *type*, das den Datentyp repräsentiert. Die möglichen Werte für *type* werden durch die Konstanten *TYPE_VOID*, *TYPE_INT32*, *TYPE_FLOAT* und *TYPE_STRING* festgelegt.

Das zweite Element mit der Bezeichnung *val* enthält den eigentlichen Wert. Da die Interpretation dieses Elements vom Femtoscript-Typ des Wertes abhängig sein muss, ist es als *union*-Datentyp *union UntypedValue* realisiert, der einen Wert vom Typ *int32_t*, *float* oder *struct StringVar* enthalten kann.

Ein Vorteil dieser Konstruktion liegt darin, dass ein Femtoscript-Wert unabhängig von seinem Typ immer den gleichen Platz im Speicher belegt. Jedes der drei Elemente

[2]String-Variablen bilden hier eine Ausnahme, da der Speicher für Strings separat verwaltet wird. Strings werden später genauer beschrieben.

```
1 struct StringVar {
2     uint8_t length;
3     union {
4         struct AllocatedString *pointer;
5         char data[3];
6     } content;
7 };
```

Abbildung 5: Deklaration von struct StringVar aus der Headerdatei oestrings.h

im *union UntypedValue* belegt mit dem verwendeten Compiler (avr-gcc mit der Option -fpack-struct) vier Bytes, mit dem Typbyte zusammen nimmt ein Femtoscript-Wert auf dem ATmega also fünf Bytes im Speicher ein.

5.4.2. Strings

Weil Strings eine variable Länge besitzen und deutlich mehr Speicher benötigen können als die anderen Datentypen, werden sie gesondert behandelt. Für das Arbeiten mit Stringwerten ist der Quellcode in den Dateien *oestrings.h* und *oestrings.c* verantwortlich, der Funktionen und Makros zum Erzeugen, Bearbeiten und Löschen von Strings enthält.

Der im *union UntypedValue* enthaltene *struct StringVar* enthält entweder eine Referenz auf einen getrennt abgelegten String, oder direkt einen String mit einer Länge von bis zu zwei Zeichen. Auf diese Weise muss für kurze Strings kein zusätzlicher Speicher reserviert und verwaltet werden. Dadurch ist das Arbeiten mit Strings aus einzelnen Zeichen effizient möglich.

Die Verwendung von Referenzen hat den Vorteil, dass ein langer String bei der Ausdruckauswertung und der Parameterübergabe nicht mehrmals kopiert werden muss, was in erster Linie Arbeitsspeicher spart. In diesen Fällen wird einfach eine weitere Referenz auf den gleichen String erzeugt. Um unerwünschte Seiteneffekte zu vermeiden, die entstehen könnten, wenn mehrere Variablen auf den selben String verweisen, werden Stringvariablen als unveränderlich behandelt. Funktionen, die Strings bearbeiten, erzeugen also immer eine neue Kopie und lassen den alten String unverändert.

Das erste Element im *struct StringVar* ist das Byte *length*, das die Anzahl der Zeichen im String angibt. Bei kurzen Strings mit bis zu zwei Zeichen folgen darauf direkt die Zeichen des Strings, gefolgt von einem Nullbyte. Längere Strings werden in einem anderen Speicherbereich abgelegt, der mit Hilfe von *malloc()* reserviert wird. In diesem Fall enthält StringVar neben der Länge nur einen Zeiger auf den reservierten Speicherbereich, der die Datenstrukur *struct AllocatedString* enthält.

Neben den eigentlichen Zeichendaten enthält ein *AllocatedString* auch einen Referenzzähler (*refCount*). Dieser wird jedes mal erhöht, wenn durch das Kopieren einer bestehenden Referenz eine neue Referenz auf diesen *AllocatedString* angelegt wird, und jedes mal verringert, wenn eine Referenz auf diesen *AllocatedString* gelöscht wird. Wenn der Referenzzähler auf Null verringert wird, wird der Speicher des *AllocatedString* wieder

```
1 struct AllocatedString {
2     struct AllocatedString *nextString;
3     uint8_t refCount;
4     char data[];
5 };
```

Abbildung 6: Deklaration von struct AllocatedString aus der Headerdatei oestrings.h

freigegeben.

Das korrekte Verwalten der Referenzzähler bringt einen gewissen Aufwand mit sich. Jedes mal, wenn ein Femtoscript-Wert kopiert wird, muss geprüft werden, ob es sich um einen String handelt, und falls nötig der Referenzzähler erhöht werden. Noch etwas schwieriger ist es, sicherzustellen, dass der Referenzzähler immer korrekt verringert wird, da Femtoscript-Werte auf verschiedene Arten gelöscht werden können, zum Beispiel indem sie mit einem neuen Wert überschrieben werden, oder indem sie als lokale Variablen freigegeben werden, sobald von der entsprechenden Funktion zurückgesprungen wird. Tatsächlich werden alle lokalen Variablen vor dem Rücksprung auf Stringreferenzen geprüft, die dann freigegeben werden.

Im Fehlerfall ist es dagegen manchmal nicht mehr ohne Schwierigkeiten möglich, den Interpreterstack nach freizugebenden Stringreferenzen zu durchsuchen, bevor der Interpreter beendet wird. Es sollte aber auch bei Fehlern im interpretierten Programm sichergestellt sein, dass der Interpreter allen reservierten Speicher wieder freigibt. Als Lösung für dieses Problem sind alle *AllocatedStrings* über das Feld *nextString* zu einer verketteten Liste verbunden. Die globale Variable *firstString* enthält einen Zeiger auf den ersten *AllocatedString*, dieser verweist über das Feld *nextString* auf den nächsten. Die Kette endet mit dem Nullzeiger. Beim Beenden des Interpreters im Fehlerfall muss also lediglich diese verkettete Liste traversiert werden, um alle für Strings reservierten Speicherbereiche zu finden und freizugeben.

5.4.3. Variablen

Dieser Abschnitt beschreibt zunächst, wie Variablen auf dem Interpreterstack abgelegt werden. Danach werden die internen Abläufe bei Zugriff und Deklaration erläutert.

Globale Variablen werden in der Variablenliste im globalen Rahmen abgelegt, lokale Variablen befinden sich in der Variablenliste des lokalen Rahmens des entsprechenden Funktionsaufrufs. Der Unterschied zwischen beiden Fällen liegt aber nur im Speicherort, das Format der Variablenlisten ist in beiden Fällen gleich.

Vor der eigentlichen Variablenliste befindet sich ein Zeiger, der auf das erste Byte hinter der Variablenliste zeigt. Dieser Zeiger dient zwei unterschiedlichen Zwecken: Zum einen wird er verwendet, um beim Durchsuchen der Variablen das Ende der Variablenliste zu erkennen, zum anderen gibt er die nächste Position an, an die eine neue Variable zur Liste hinzugefügt werden kann. Der zweite Aspekt wird im Absatz über die Deklaration genauer ausgeführt.

Auf den Zeiger folgt eine alternierende Liste aus Bezeichnern und Variablenwerten. Für einen Bezeichner ist dabei zuerst dessen Länge als Bytewert abgelegt, darauf folgen die eigentlichen Zeichen des Variablennamens. Daran schließt sich der Variablenwert an, der als *struct TypedValue* gespeichert ist.

Wenn der Interpreter auf eine Variable zugreifen will, entweder um ihren Wert zu bestimmen oder um einen neuen Wert zuzuweisen, durchsucht er zunächst die lokale Variablenliste. Dazu wird jeder Bezeichner in der Variablenliste mit dem Bezeichner der gewünschten Variable im Quelltext verglichen, bis der richtige Eintrag in der Liste gefunden wurde. Falls die Variable nicht in der lokalen Liste gefunden wird, durchsucht der Interpreter als nächstes die globale Liste auf die gleiche Weise. Falls der Bezeichner aus dem Quelltext hier auch nicht enthalten ist, wird das Programm mit dem Fehler „Unbekannte Variable“ abgebrochen. Wenn der Bezeichner gefunden wurde, kann der dazugehörige Wert gelesen oder geändert werden.

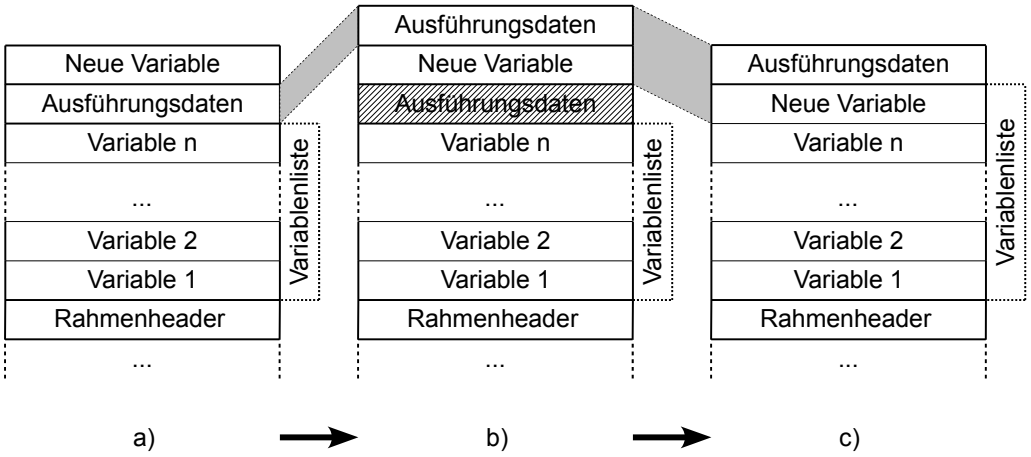

Abbildung 7: Vorgehen des Interpreters bei der Deklaration einer neuen Variable. Die Diagramme zeigen das obere Ende des Interpreterstacks. Die unterste Zeile beginnt in allen drei Zuständen an der gleichen Speicheradresse. Die grauen Verbindungen verdeutlichen Kopier-/Verschiebevorgänge.

Bei der Deklaration einer Variable muss ihr Bezeichner zusammen mit einem Initialwert an die entsprechende Variablenliste angehängt werden. Die Deklarationen werden vom Interpreter ähnlich wie Anweisungen in der Hauptschleife bearbeitet. Der Grund dafür liegt darin, dass Femtoscript die Initialisierung von Variablen bei der Deklaration erlaubt. Im Bereich der Variablendeklarationen können daher Funktionsaufrufe auftauchen, in denen wiederum Variablen deklariert und initialisiert werden. Aufgrund der Überlegungen zur Rekursionsvermeidung in Abschnitt 5.3.2 müssen Deklarationen da-

her in der Hauptschleife verarbeitet werden. Das hat wiederum zur Folge, dass sich bei der Auswertung der Deklarationen auf dem Interpreterstack bereits Ausführungsdaten über der Variablenliste befinden.

Abbildung 7 illustriert die Veränderungen auf dem Interpreterstack beim Anlegen einer neuen Variable. Zunächst schreibt der Interpreter den neuen Eintrag für die Variablenliste (Bezeichner und Void-Wert) oben auf den Stack (Zustand a). Im zweiten Schritt werden die Ausführungsdaten, die sich zwischen dem Ende der Variablenliste und dem Beginn der neuen Variable befinden über den neuen Eintrag kopiert (Zustand b). Daraufhin wird die neue Variable zusammen mit den kopierten Ausführungsdaten an das obere Ende der Variablenliste verschoben. Als letztes wird der Zeiger angepasst, der das Ende der Variablenliste markiert (Zustand c). Damit ist die neue Variable fertig angelegt. Die Initialisierung wird daraufhin wie eine normale Zuweisung behandelt.

5.5. Ausdrücke

Ausdrücke werden vom Interpreter nach einem Verfahren ausgewertet, das nach dem Vorbild des Shunting-yard Algorithmus entwickelt wurde. Dabei muss ein Ausdruck nur einmal von links nach rechts gelesen werden, um das Ergebnis zu erhalten.

Wenn man einen Ausdruck von links nach rechts liest, kann man aufgrund der Operatorprioritäten oft nicht alle Operationen sofort ausführen. Auch die Klammerung eines Teilausdrucks kann eine verzögerte Auswertung nötig machen. Im Beispiel 1+2*3 kann die Addition nicht sofort ausgeführt werden, weil zuerst der Wert des rechten Operanden (2*3) berechnet werden muss. Operatoren, die noch nicht angewendet werden können, werden daher zunächst zusammen mit ihrem linken Operanden auf dem Interpreterstack abgelegt und erst dann ausgeführt, wenn ihr rechter Operand vollständig ausgewertet ist.

Im eben genannten einfachen Beispiel würde also der Operand 1 zusammen mit dem Operator + auf den Stack abgelegt, da ein Operator mit höherer Priorität folgt und die Addition noch nicht angewendet werden kann. Darauf folgt die Multiplikation 2*3, die sofort ausgerechnet und durch das Ergebnis 6 ersetzt werden kann. Jetzt kann die auf dem Interpreterstack gespeicherte Addition ausgeführt werden.

Wenn man mit diesem Verfahren längere Ausdrücke bearbeitet, enthält der Stack (von unten nach oben betrachtet) immer eine Liste von Operatoren mit steigender Priorität. Wenn ein neuer Operator (*rightOp*) gelesen wird, wird die Priorität von *rightOp* mit der des obersten Operators auf dem Stack (*leftOp*) verglichen. Falls die Priorität von *leftOp* größer oder gleich ist, wird *leftOp* vom Stack genommen und angewendet. Der neue oberste Operator auf dem Stack wird zu *leftOp* und der Vorgang wird wiederholt. Sobald *leftOp* eine kleinere Priorität besitzt als *rightOp* wird *rightOp* auf den Stack gelegt und ein neuer Operator wird gelesen.

Diese Beschreibung übergeht allerdings noch die Behandlung der Operanden. Wenn man diese hinzufügt, erhält man den Algorithmus, der in Abbildung 8 gezeigt ist.

Dieser Vorgang wird wiederholt, bis der gesamte Ausdruck ausgewertet ist. Damit dieses Verfahren das Ende des Ausdrucks erkennen kann, wird vorher ein spezieller Operator auf dem Stack abgelegt, der eine niedrigere Priorität besitzt als alle normal in

1. Lese einen Operanden und speichere seinen Wert in *accumulator*.
2. Lese einen Operator (*rightOp*).
3. Solange die Priorität des obersten Operators auf dem Stack gleich oder höher ist als die von *rightOp*:
   a) Nimm den obersten Operator vom Stack (*leftOp*).
   b) Nimm den linken Operanden von *leftOp* vom Stack (*leftOperand*).
   c) Wende *leftOp* auf *leftOperand* und *accumulator* an und speichere das Ergebnis in *accumulator*.
4. Lege *accumulator* (als linken Operanden von *rightOp*) auf den Stack.
5. Lege *rightOp* auf den Stack.

Abbildung 8: Ein vereinfachter Algorithmus zur Ausdrucksauswertung. Das Ende des Ausdrucks, unäre Operatoren und Klammern werden noch nicht behandelt.

einem Ausdruck vorkommenden Operatoren. Wenn beim Lesen von *rightOp* kein gültiger Operator gefunden wird, erhält *rightOp* ebenfalls diesen Spezialoperator zugewiesen, um das Ende des Ausdrucks zu signalisieren. Damit haben alle restlichen Operatoren auf dem Stack eine höhere Priorität als *rightOp* und werden in der Schleife abgearbeitet, bis der Spezialoperator vom Stack gelesen wird. Dieser beendet die Ausdrucksauswertung, da jetzt alle Operationen ausgeführt wurden und *accumulator* als Ergebnis den Wert des Ausdrucks enthält.

Als zusätzlicher Spezialfall müssen unäre Operatoren behandelt werden. Da diese keinen linken Operanden besitzen, tauchen sie beim Lesen des Ausdrucks an den Stellen auf, an denen ein Operand erwartet würde. Beim Ausführen muss darauf geachtet werden, dass für diese Operatoren kein linker Operand auf dem Stack liegt.

Wenn man das Verfahren gemäß dieser Überlegungen anpasst, erhält man den Algorithmus aus Abbildung 9.

Die öffnende und schließende Klammer werden ebenfalls als Spezialoperatoren niedrigster Priorität behandelt. Auf diese Weise wird der Klammerinhalt ähnlich ausgewertet wie ein eigener Ausdruck, mit dem Unterschied, dass öffnende und schließende Klammern im Ausdruck gezählt werden. So können Fehler in der Klammerung erkannt und gemeldet werden.

5.6. Funktionen

Ein weiterer Aspekt von Femtoscript sind Funktionen. Dieser Abschnitt beschreibt genauer, wie Funktionen aufgerufen werden. Das Verarbeiten der Funktionsdefinitionen wurde Bereits in Abschnitt 5.3.1 erläutert.

Der Aufruf einer Funktion erfolgt entweder aus einem Ausdruck heraus, wobei der

1. Lege den Ausdruck-Ende-Operator auf dem Stack ab.
2. Solange der Ausdruck nicht fertig ausgewertet ist:
   a) Lese unäre Operatoren und lege sie auf dem Stack ab, bis kein gültiger unärer Operator mehr gefunden wird.
   b) Lese einen Operanden und speichere seinen Wert in *accumulator*.
   c) Lese einen Operator (*rightOp*).
   d) Solange die Priorität des obersten Operators auf dem Stack gleich oder höher ist als die von *rightOp*:
      i. Nimm den obersten Operator vom Stack (*leftOp*).
      ii. Falls *leftOp* ein unärer Operator ist:
         A. Falls *leftOp* der Ausdruck-Ende-Operator ist, beende die Ausdrucksauswertung.
         B. Wende *leftOp* auf *accumulator* an und speichere das Ergebnis in *accumulator*.
      iii. Sonst:
         A. Nimm den linken Operanden von *leftOp* vom Stack (*leftOperand*).
         B. Wende *leftOp* auf *leftOperand* und *accumulator* an und speichere das Ergebnis in *accumulator*.
   e) Lege *accumulator* (als linken Operanden von *rightOp*) auf den Stack.
   f) Lege *rightOp* auf den Stack.

Abbildung 9: Erweiterter Algorithmus zur Ausdrucksauswertung.

Funktionsaufruf als Operand betrachtet wird, oder als eigene Anweisung. In beiden Fällen muss vor dem eigentlichen Funktionsaufruf die Liste der Aufrufparameter ausgewertet werden. Jeder Parameter in dieser Liste wird zunächst als eigener Ausdruck ausgewertet, der Ergebniswert wird auf dem Interpreterstack abgelegt. Zusätzlich wird auch ein Verweis auf den Namen der Funktion und die Anzahl der bisher ausgewerteten Aufrufparameter gespeichert.

Wenn alle Aufrufparameter ausgewertet sind, wird versucht, die Funktion aufzurufen. Als Erstes wird eine eingebaute Funktion mit dem angegebenen Bezeichner gesucht. Falls diese nicht gefunden wird, wird die Funktionsliste auf dem Interpreterstack durchsucht, um die Quelltextposition der richtigen benutzerdefinierten Funktion zu finden. Falls auch hier der angegebene Bezeichner nicht gefunden wird, wird der Fehler „Unbekannte Funktion“ ausgelöst und der Interpreter wird beendet.

Hier wird zunächst auf die Behandlung von benutzerdefinierten Funktionen eingegangen, der Aufruf von eingebauten Funktionen wird im nächsten Abschnitt genauer beschrieben.

Zum Ausführen einer benutzerdefinierten Funktion sind mehrere Schritte nötig. Um den späteren Rücksprung aus der Funktion zu ermöglichen, müssen die aktuelle Position im Quelltext und der aktuelle Zeiger auf den Beginn des lokalen Rahmens zwischengespeichert werden. Zusätzlich müssen die Aufrufparameter auf dem Interpreterstack in eine lokale Variablenliste für die neu aufgerufene Funktion umgewandelt werden. Diese Elemente bilden zusammen den neuen lokalen Rahmen für die aufgerufene Funktion. Die Rücksprungadresse, der Zeiger auf den alten lokalen Rahmen und ein Zeiger auf das Ende der lokalen Variablenliste werden zusammen als Rahmenheader bezeichnet, direkt dahinter muss die Variablenliste folgen.

Die Aufrufparameter liegen auf dem Interpreterstack jedoch nur als Werte ohne Bezeichner vor (Abbildung 10, Zustand (a)). Daher besteht der erste Schritt darin, die Parameterliste der Funktion einzulesen und dabei jeden Bezeichner oben auf dem Interpreterstack abzulegen. Hinter jeden Bezeichner wird der entsprechende Aufrufparameter kopiert. Danach befindet sich der Interpreterstack im Zustand (b).

Die alte Liste mit den Werten der Aufrufparameter wird jetzt nicht mehr benötigt (im Diagramm durch Schraffur gekennzeichnet). Daher wird der neue lokale Rahmen an der alten Startadresse dieser Liste auf dem Interpreterstack begonnen. Vorher muss allerdings noch die neue Liste, die auch die Bezeichner der Parameter enthält, an die richtige Position auf dem Interpreterstack verschoben werden. Die neue Position wird so berechnet, dass zwischen der Startadresse des neuen lokalen Rahmens und dem neuen Beginn der Liste genau soviel Platz gelassen wird, wie für den noch zu erstellenden Rahmenheader benötigt wird.

Das Befüllen des Rahmenheaders ist der letzte Schritt. Im Rahmenheader werden die Quelltextposition für den Rücksprung, der Zeiger auf den lokalen Rahmen der aufrufenden Funktion und der Zeiger auf das Ende der lokalen Variablenliste abgelegt. Damit entspricht der Aufbau auf dem Stack jetzt dem Zustand (c) in Abbildung 10, und der Interpreter kann mit dem Ausführen der Funktion beginnen, angefangen mit den Deklarationen für die lokalen Variablen.

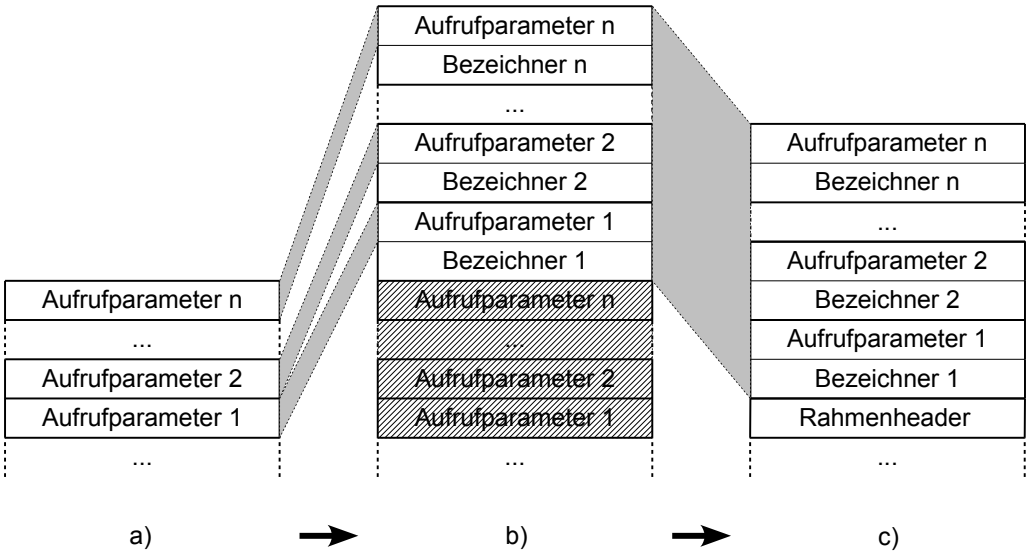

Abbildung 10: Erstellen eines lokalen Stackrahmens aus der Liste der Aufrufparameter. Die Diagramme zeigen das obere Ende des Interpreterstacks. Die unterste Zeile beginnt in allen drei Zuständen an der gleichen Speicheradresse. Die grauen Verbindungen verdeutlichen Kopier-/Verschiebevorgänge.

5.6.1. Eingebaute Funktionen

Um zu prüfen, ob ein Bezeichner zu einer eingebauten Funktion gehört und diese gegebenenfalls auszuführen, ruft der Interpreter *callBuiltinFunction()* aus der Datei *builtinFunctioncall.gen.c* auf. Falls der Name der aufgerufenen Femtoscript-Funktion dem Namen einer eingebauten Funktion entspricht, wird durch *callBuiltinFunction()* die C-Funktion aufgerufen, in der die eingebaute Funktion implementiert ist. Ansonsten meldet *callBuiltinFunction()* dem Interpreter, dass keine passende Funktion gefunden wurde, und der Interpreter sucht den Namen der aufgerufenen Funktion in der Liste der benutzerdefinierten Funktionen.

Femtoscript kann um neue eingebaute Funktionen erweitert werden. Das kann nützlich sein, um bestimmte Hardwareressourcen wie eine serielle Schnittstelle oder Soundausgabe zu unterstützen, oder um oft benötigte Berechnungen komfortabler und effizienter umzusetzen. Man kann auch nicht benötigte Funktionen entfernen, um Speicherplatz zu sparen. Die nötigen Änderungen am Quelltext werden hier aufgrund des starken thematischen Zusammenhangs zusammen mit detaillierteren Informationen zur Behandlung eingebauter Funktionen erläutert.

Um eine neue eingebaute Funktion zum Interpreter hinzuzufügen sind zwei Schritte notwendig: Erstens muss *callBuiltinFunction()* so angepasst werden, dass der Name der neuen Funktion erkannt wird. Zweitens muss eine C-Funktion mit der Implementierung

der neuen Femtoscript-Funktion erstellt werden. Beide Schritte werden im Folgenden näher beschrieben.

callBuiltinFunction() berechnet zuerst einen Hashwert des Funktionsnamens, und prüft diesen dann in einem switch-Statement gegen die vorberechneten Hashwerte der Namen der eingebauten Funktionen. Da die Namen der meisten eingebauten Funktionen unterschiedliche Hashwerte haben, ist so in den meisten Fällen nur noch ein einziger Stringvergleich nötig, bevor die richtige C-Funktion aufgerufen werden kann.

Um eine neue Funktion in Femto einzubauen, muss *callBuiltinFunction()* so erweitert werden, dass der Bezeichner der neuen Funktion erkannt wird und zum Aufruf der richtigen C-Funktion führt. Durch die Verwendung des oben beschriebenen Hash-Verfahrens wird das Hinzufügen und Entfernen von eingebauten Funktionen allerdings erschwert. Um diesen Aufwand zu vermeiden, wird *callBuiltinFunction()* nicht von Hand bearbeitet. Statt dessen wird die Datei *builtinFunctioncall.gen.c*, die diese Funktion enthält, von einem Codegenerator erzeugt, der im Projektverzeichnis unter *tools/builtinFunctionCodeGenerator.html* zu finden ist. Der Generator ist in Javascript geschrieben und sollte in allen üblichen Browsern funktionieren.

Der Generator nimmt als Eingabe die Namen der eingebauten Funktionen entgegen, die in ein Eingabefeld im oberen Teil der HTML-Seite eingegeben werden. Jeder Name steht dabei in einer eigenen Zeile. Die Datei *builtinFunctionNames.txt*, die sich ebenfalls im Verzeichnis *tools* befindet, enthält die Namen aller momentan in Femtoscript vorhandenen eingebauten Funktionen und kann als Vorlage in das Eingabefeld kopiert werden.

Als weitere Einstellmöglichkeit kann die Anzahl der zu verwendenden Hash-Buckets vorgegeben werden. Eine kleinere Anzahl von Buckets verteilt die Funktionsnamen auf weniger case-Statements, in denen dann aber zwischen mehr möglichen Funktionsnamen unterschieden werden muss. Im Allgemeinen ist es allerdings nicht empfehlenswert, die vorgegebene Zahl von 256 Buckets zu ändern, da eine zu große Zahl von Buckets fast keine Leistungsnachteile bringt, eine zu niedrige dagegen schon.

Durch einen Klick auf die entsprechende Schaltfläche wird nun in das untere Textfeld der neue Programmcode generiert, der in die Datei *builtinFunctioncall.gen.c* kopiert werden kann.

Die entsprechenden C-Funktionen zu den Femtoscript-Funktionen erhalten ihre Namen nach einem festen Schema aus den Namen der entsprechenden Femtoscript-Funktionen. Der Name beginnt immer mit "builtin", gefolgt vom Femtoscript-Funktionsnamen. Der erste Buchstabe des Femtoscript-Namen wird allerdings in Großschreibung überführt. Die C-Funktion, die die Femtoscript-Funktion *print()* implementiert, heißt also *builtinPrint()*. Diese Funktionen sind in der Datei *builtinFunctions.h* deklariert und in der Datei *builtinFunctions.c* definiert. Es ist wichtig, diese Namenskonvention einzuhalten, da der Codegenerator die Funktionsaufrufe nach diesem Schema herleitet.

Die C-Funktionen geben keinen Wert zurück und übernehmen nur einen Parameter: Einen Zeiger auf einen Integer, der die Anzahl der Parameter enthält, die der

Femtoscript-Funktion übergeben wurden. Die eigentlichen Femtoscript-Parameter werden mit Hilfe der Funktionen *popIntParam()*, *popFloatParam()*, *popStringParam()* und *popMixedParam()* abgefragt, die in der Datei *builtinFunctions.c* definiert sind. Jede dieser Funktionen nimmt einen Parameter vom Stapel und übergibt seinen Wert an den Aufrufer. Der letzte Parameter wird dabei zuerst zurückgegeben.

Um sicherzustellen, dass die richtige Anzahl an Parametern vom Stapel genommen wird, muss allen popParam-Funktionen der Zeiger auf die Anzahl der Parameter übergeben werden. Der Wert wird bei jedem popParam-Aufruf um eins verringert. Falls eine builtin-Funktion versucht, mehr Parameter abzuheben als sich auf dem Stack befinden, wird der Interpreter automatisch mit dem Fehlercode *ERROR_NUM_PARAMETERS* beendet. Der gleiche Fehler wird ausgelöst, wenn sich nach Ausführung der builtin-Funktion noch Parameter auf dem Stack befinden.

popIntParam() gibt einen Wert vom Typ *int32_t* zurück, falls der Parameter den Femtoscript-Typ Integer hat. Ansonsten wird der Interpreter mit einem Typfehler beendet. *popFloatParam()* gibt dagegen einen *float*-Wert zurück, sofern der Parameter vom Typ Integer oder Float ist, und löst ansonsten ebenfalls einen Typfehler aus.

popStringParam() gibt eine Stringvariable zurück. Eine automatische Umwandlung anderer Typen in String findet nicht statt; auch hier wird ein Typfehler ausgelöst, falls der Femtoscript-Funktion nicht der richtige Datentyp übergeben wurde. Die Stringvariable hat den Typ *struct StringVar*, der für die interne Verwaltung von Strings im Interpreter verwendet wird. Um einen Zeiger auf die eigentlichen Stringdaten zu erhalten, die als nullterminierter C-String abgelegt sind, übergibt man die Adresse der Stringvariablen an die Funktion *getStringDataPointer()*, die in *oestrings.h* definiert ist.

Jede vom Stack gelesene Stringvariable, die nicht weiter verwendet wird, muss nach der Verarbeitung freigegeben werden, um Speicherlecks zu vermeiden. Zu diesem Zweck wird die Funktion *releaseString()* aufgerufen, die ebenfalls in der Datei *oestrings.h* definiert ist und die Adresse der Stringvariable als Parameter erwartet. Falls die Stringvariable weiter benutzt wird, z.B. indem sie als Rückgabewert der Funktion verwendet wird, darf sie dagegen nicht freigegeben werden.

popMixedParam() liefert den Parameter als *struct TypedValue* zurück. Dieser Datentyp enthält einen beliebigen Femtoscript-Wert inklusive Typinformation. Damit ist *popMixedParam()* die allgemeinste Funktion, um einen Parameter zu erhalten.

Die Funktion *clearTypedValue()* aus der Datei *variables.h* löscht den Inhalt einer TypedValue und setzt ihren Typ auf Void. Falls die TypedValue eine Stringvariable enthält, wird diese dabei freigegeben. *clearTypedValue()* sollte daher immer angewendet werden, wenn eine TypedValue nicht mehr benötigt wird und einen Stringwert enthalten könnte.

Um einen Rückgabewert aus der eingebauten Funktion zu liefern, schreibt man das Ergebnis der Funktion in die globale Variable *accumulator*, die ebenfalls den Typ *struct TypedValue* besitzt. Vor dem Aufruf der builtin-Funktionen wird *accumulator* gelöscht und erhält den Typ Void zugewiesen, so dass keine weitere Arbeit notwendig ist, falls die entsprechende Femtoscript-Funktion Void zurückgeben soll.

Ein Beispiel für eine neue eingebaute Funktion könnte die Funktion *max()* sein, die zwei Integer-Parameter übernimmt und den größeren davon zurückgibt.

Dazu ruft man zuerst den oben beschriebenen Codegenerator auf und kopiert die Liste der aktuell definierten Funktionen in das linke obere Textfeld. Diese Liste wird um eine neue Zeile mit dem Namen der Funktion *max* ergänzt, und der Generator wird gestartet. Das Ergebnis ersetzt den alten Inhalt der Datei *callBuiltinFunction()*.

Jetzt ergänzt man *builtinFunctions.h* um die Deklaration der neuen C-Funktion:

```
1 void builtinMax(uint8_t *numParams);
```

In *builtinFunctions.c* muss als letztes noch die tatsächliche Implementierung der Funktion eingetragen werden:

```
1  void builtinMax(uint8_t *numParams) {
2      int32_t param2 = popIntParam(numParams);
3      int32_t param1 = popIntParam(numParams);
4      accumulator.type = TYPE_INT32;
5      if(param2>param1) {
6          accumulator.val.i32 = param2;
7      } else {
8          accumulator.val.i32 = param1;
9      }
10 }
```

6. Texteditor und Hilfsbibliotheken

6.1. ATmega-Texteditor

Die Entwicklung eines leistungsfähigen und allgemein verwendbaren Texteditors ist eine komplizierte Aufgabe, die über den Umfang dieser Arbeit hinausgehen würde. Der Editor, der im Rahmen dieser Arbeit für den Einchipcomputer entwickelt wurde, verwendet daher einen sehr einfachen Ansatz, ist aber nur für das Bearbeiten kleiner Programme geeignet, die vollständig in den Arbeitsspeicher des ATmega passen.

6.1.1. Start und Hauptschleife

Der Editor dient als Hauptprogramm der Programmierumgebung und enthält daher die *main()*-Methode. Hier werden zunächst die Video- und die Tastaturbibliothek initialisiert. Danach wird ein leerer Textpuffer angelegt, und der Bildschirmspeicher wird aktualisiert. Da der Textpuffer leer ist, wird dabei ein leerer Bildschirm mit dem Cursor in der oberen linken Ecke erzeugt.

Nun beginnt die Ausführung der Hauptschleife. Hier wird in jedem Durchlauf zuerst so lange die Tastatur abgefragt, bis eine Taste gedrückt wird. Die Eingabe wird verarbeitet, indem eine passende Funktion aufgerufen wird. Ein Druck auf die Taste „Ende“

löst beispielsweise den Aufruf der Funktion *moveCursorToEndOfLine()* aus, die Eingabe eines druckbaren Zeichens führt zum Aufruf von *insertCharacter(char c)* mit dem eingegebenen Zeichen als Parameter. F5 startet den Interpreter durch Aufruf von *interpretProgram()*. Nach der Ausführung wird der Rückgabewert ausgewertet und eine entsprechende Erfolgs- oder Fehlermeldung ausgegeben.

Die Hauptschleife des Editors ist eine Endlosschleife und wird nie beendet.

6.1.2. Textpuffer

Der Textpuffer ist ein dynamisch allozierter Speicherbereich, der die Anzahl der Zeichen im Text, die aktuelle Cursorposition sowie die eigentlichen Textdaten enthält. Der Puffer wird so angelegt, dass er die gleiche Größe besitzt wie der EEPROM-Speicher des ATmega. Damit ist es einfach möglich, den gesamten Puffer abzuspeichern und wieder zu laden.

Der Text ist immer zusammenhängend abgelegt und wird durch ein Nullbyte terminiert. Wenn ein Zeichen im Text eingefügt oder gelöscht wird, wird der gesamte nachfolgende Text im Speicher verschoben. Diese Technik wurde gewählt, weil sie einfach zu implementieren und zu verstehen ist, und bei der Größe der bearbeiteten Texte keine merklichen Geschwindigkeitsprobleme aufweist. Für das Bearbeiten größerer Texte wären kompliziertere Techniken und Datenstrukturen nötig, die ein Bearbeiten von Texten ermöglichen, die sich nur teilweise im Arbeitsspeicher und teilweise auf einem externen Speicher befinden.

Die Quellcodeverwaltung für den Interpreter ist so realisiert, dass der gesamte Text als ein Abschnitt an den Interpreter übergeben wird. Damit ist ein Nachladen während der Programmausführung nicht notwendig. Es wäre auch denkbar, den Quelltext vor der Ausführung immer im EEPROM abzuspeichern und dem Interpreter zeilenweise bereitzustellen. Dadurch könnte dem Interpreter deutlich mehr Speicher für die Ausführung zur Verfügung gestellt werden.

6.2. VGA-Videoausgabe

6.2.1. Das VGA-Videosignal

Der VGA-Standard ist auf den Betrieb von Röhrenmonitoren ausgelegt. Das VGA-Videosignal wird daher über eine HSYNC- und eine VSYNC-Leitung für die Synchronisation der horizontalen und vertikalen Rücksprünge des Elektronenstrahls, sowie über drei Leitungen für die Intensität der drei Farbkomponenten Rot, Grün und Blau an der aktuellen Position des Elektronenstrahls übertragen. Es existieren mehrere Standard-Videomodi, die von den meisten Monitoren unterstützt werden und die die Anzahl der Bildzeilen, die Frequenzen und Polarisationen der Synchronisationspulse und weitere Details des Signals festlegen.

6.2.2. Hardwareanbindung an den ATmega

Auf der verwendeten Platine sind die drei Farbleitungen miteinander verbunden, und es wird nur ein digitaler Ausgang zur Steuerung dieser Leitungen benutzt. Dadurch ist nur die Ausgabe eines Schwarzweiß-Bildes möglich.

6.2.3. Erzeugung der Sync-Pulse

Um eine möglichst große Regelmäßigkeit der horizontalen Synchronisationspulse zu erreichen, ist es günstig, einen PWM-Generator zur Erzeugung dieses Signals zu verwenden. Dadurch wird ohne weiteren Programmieraufwand eine hohe Stabilität des HSYNC-Signals gewährleistet. Auf dem verwendeten ATmega644 kann hierzu der Timer 0 eingesetzt werden, auf kleineren Modellen der ATmega-Serie verfügt dieser Timer jedoch nicht über die nötige Funktionalität. Timer 1 ist auf allen ATmega-Modellen für diese Aufgabe geeignet, ist allerdings als der einzige verfügbare 16-Bit Timer eine nützliche Hardwareressource, die für andere Aufgaben gebraucht werden könnte.

Der Überlauf-Interrupt des Timer 0 wird verwendet, um einen Zähler für die aktuelle Videozeile zu erhöhen und bei bestimmten Zeilen zusätzliche Aufgaben zu erfüllen. In Zeile 0 wird so die VSYNC-Leitung invertiert, um den Beginn des VSYNC-Pulses auszulösen. In einer späteren Zeile wird die Leitung wieder invertiert, um den Puls zu beenden. Die Länge des VSYNC-Pulses ist vom Videomodus abhängig.

6.2.4. Ausgabe des sichtbaren Bildes

Eine weitere besondere Zeile ist die, in der der sichtbare Bildbereich beginnt. Ab diesem Zeitpunkt ist der Prozessor fast dauerhaft damit beschäftigt, den Bildschirminhalt auf die VGA-Leitungen auszugeben. Zwischen den Zeilen existieren zwar kurze Pausen, diese sind jedoch nicht ausreichend lang um die Interruptroutine zu verlassen und bis zum Beginn der nächsten Zeile wieder zur Ausgabe bereit zu sein. Die überschüssigen Zyklen könnten aber für andere Aufgaben verwendet werden, zum Beispiel zur Umsetzung eines Synthesizers zur Audioausgabe.

Die Videoschleife gibt einen Bildschirmspeicher mit Textzeichen aus. Dazu werden in jeder Videozeile die entsprechenden Zeichen aus dem Bildschirmspeicher gelesen und in einer Fonttabelle nachgeschlagen, welche die Grafiken der einzelnen Zeichen als Bitmuster enthält. Ein gesetztes Bit steht dabei für einen weißen und ein gelöschtes für einen schwarzen Pixel.

Eine Textzeile besteht aus mehreren Videozeilen. Daher werden in mehreren aufeinander folgenden Videozeilen wiederholt die gleichen Zeichen gelesen, aber in einer anderen Zeile der Fonttabelle nachgeschlagen. So werden von oben nach unten die Zeichengrafiken aufgebaut. Wenn das untere Ende einer Textzeile erreicht ist, wird in der folgenden Videozeile die nächste Textzeile begonnen.

6.3. PS/2-Tastatureingabe

6.3.1. Beschreibung des PS/2-Tastaturprotokolls

PS/2-Tastaturen benutzen zur Datenübertragung ein synchrones serielles Protokoll mit einer Daten- und einer Clockleitung. Der Übertragungsrahmen besteht aus einem Startbit, gefolgt von acht Datenbits, einem ungeraden Paritätsbit und einem Stoppbit, wobei der Takt von der Tastatur erzeugt wird. Neben der Datenübertragung in diesem Format hat das Gerät, an das die Tastatur angeschlossen ist die Möglichkeit, durch Beeinflussung der beiden Leitungen seinen aktuellen Zustand (empfangsbereit, sendebereit, beschäftigt) an die Tastatur zu übertragen oder die Tastatur zurückzusetzen.

Nach dem Einschalten sendet die Tastatur zunächst ein Byte mit dem Wert 0xAA, um die Betriebsbereitschaft anzuzeigen. Danach wird für jeden Tastendruck (und bei jedem Lösen einer Taste, mit Ausnahme der Pause-Taste) ein Scancode gesendet, der das Ereignis beschreibt. Die Bytefolgen, die für das Drücken einer Taste stehen, werden Make-Codes genannt. Um anzuzeigen, dass eine Taste gelöst wurde, wird ein tastenspezifischer Break-Code gesendet. Wenn eine Taste längere Zeit gehalten wird, wiederholt die Tastatur automatisch den Make-Code in regelmäßigen Zeitabständen, um die wiederholte Ausgabe des gedrückten Zeichens zu erreichen.

Um einem Scancode seine Bedeutung zuzuordnen, ist es nötig, das Layout der Tastatur zu kennen. So erzeugen deutsche und amerikanische Tastaturen oft die gleichen Scancodes für Tasten, die sich an der gleichen Stelle befinden, obwohl die Beschriftung dieser Tasten unterschiedlich ist. Weiterhin existieren aus historischen Gründen drei verschiedene Sätze von Scancodes für die gleichen Tasten, zwischen denen man mit einem Befehl an die Tastatur umschalten kann.

Moderne PS/2-Tastaturen benutzen nach dem Einschalten den Scancodesatz 2. Die anderen Sätze sind für die hier beschriebene Anwendung nicht interessant und werden daher nicht weiter beschrieben. Manche der folgenden Informationen gelten nicht für Satz 1 und 3.

Scancodes besitzen eine variable Länge. Sie bestehen aus einem Byte, dem optional ein oder zwei Kontrollbytes vorausgehen können, die die Bedeutung des Codebytes beeinflussen. Scancodes ohne ein solches Präfix sind einfache Make Codes. Scancodes mit dem Präfix 0xf0 sind einfache Break Codes. Das Präfix 0xe0 zeigt einen erweiterten Make Code an, während 0xe0 0xf0 einen erweiterten Break-Code einleitet.

Die erweiterten und einfachen Codes beziehen sich auf unterschiedliche Tasten, die erweiterten Codes existieren also um die Anzahl der möglichen Tasten zu erhöhen. Break- und Make-Code einer Taste sind bis auf das zusätzliche 0xf0-Byte identisch.

Die Pausetaste bildet hier eine Ausnahme, da sie einen acht Byte langen Make Code und keinen Break Code besitzt, und das Präfix 0xe1 benutzt. Je nach Anwendungszweck kann man diese Ausnahme unberücksichtigt lassen, sofern man unbekannte Scancodes ignoriert, da der Pause-Scancode dann als ein Drücken und Lösen der linken Strg-Taste sowie der NumLock-Taste interpretiert wird.

Um den Status der Tastatur-LEDs zu beeinflussen und um Einstellungen wie die Wiederholrate für Tasten zu ändern ist es möglich, Kommandos an die Tastatur zu

senden. Wenn eine Änderung der Einstellungen nicht notwendig ist, kann die Tastatur aber auch ohne das Senden von Kommandos betrieben werden.

6.3.2. Anbindung an den ATmega

Das serielle Protokoll der PS/2-Tastatur ist kompatibel mit dem der USART-Hardware des ATmega, was den nötigen Aufwand erheblich reduziert. Neben dem Initialisieren der USART mit den richtigen Parametern liegt die Aufgabe des PS/2-Moduls damit hauptsächlich im Interpretieren der empfangenen Scancodes.

Diese Aufgabe wird in einer Interruptroutine erfüllt, die durch den Empfang eines Bytes durch die USART ausgelöst wird. Im Prinzip arbeitet diese Routine wie folgt:

Abhängig von einem Statusflag, das die bisher empfangenen Kontrollbytes signalisiert (keins, 0xe0, 0xf0 oder 0xe0 0xf0) wird entschieden, wie das empfangene Byte interpretiert wird. Handelt es sich um ein weiteres Kontrollbyte, wird der Status entsprechend geändert und die Interruptroutine wird verlassen.

Wenn ein Make- oder Break-Code vollständig empfangen wurde, wird zunächst geprüft ob es sich bei der gedrückten bzw. gelösten Taste um eine Modifikatortaste handelt, die selbst keine Aktion auslöst sondern die Bedeutung der anderen Tasten beeinflusst. Beispiele dafür sind Shift und Alt-Gr. In diesem Fall wird ein Bit-Flag für die entsprechende Taste auf den neuen Zustand gesetzt.

Make Codes von nicht-Modifikatortasten werden mit Hilfe mehrerer Tabellen in einen einbyteigen Code übersetzt, der in einen Ringpuffer geschrieben wird. Break-Codes von diesen Tasten werden ignoriert. Der verwendete Code entspricht für druckbare Zeichen ISO 8859-15 (latin-9), Steuertasten (z.B. Cursor, Enter, ...) werden in die vom ISO 8859-Standard für Steuerzeichen reservierten Bereiche abgebildet (siehe Anhang A).

Die Strg-Tasten haben eine besondere Funktion. Dem Tastaturmodul kann eine Callbackfunktion übergeben werden, die aufgerufen wird, sobald eine Strg-Tastenkombination gedrückt wird. Diese Funktion wird innerhalb der Interruptroutine aufgerufen und muss ihre Bearbeitung daher schnell beenden. In der ATmega-Programmierumgebung wird diese Funktion genutzt, um beim Drücken von Strg-c das Break-Flag des Interpreters zu setzen.

Der Tastenpuffer kann über eine Funktion *PS2_getchar()* abgefragt werden, die entweder die nächste gedrückte Taste als Wert im Bereich von *unsigned char* zurückliefert, oder *EOF* (definiert in *stdio.h*) falls der Tastenpuffer leer ist. Da der Tastenpuffer als Ringpuffer organisiert ist, geht der gesamte Pufferinhalt verloren falls mehr als sieben Zeichen unabgefragt bleiben.

7. Fazit

7.1. Verwandte Arbeiten

Als vergleichbares Projekt ist hier vor allem AVR-ChipBASIC zu erwähnen[5]. ChipBASIC ist ebenfalls eine Programmierumgebung für ATmega-MCUs, die allerdings einen

BASIC-Dialekt als Sprache verwendet und vollständig in AVR-Assember entwickelt wurde, also nicht portierbar ist.

ChipBASIC verfügt über beachtliche Funktionalität. Es existiert direkte Unterstützung für verschiedene Ein- und Ausgabemöglichkeiten wie Audioausgabe, serielle Schnittstelle und I^2C-Bus. Ein eingebauter Debugger ermöglicht das Betrachten von Variblenhalten bei einer Programmunterbrechung.

Die verwendete Sprache ist allerdings in vielerlei Hinsicht weniger komfortabel als Femtoscript. Zum Beispiel existieren genau 26 Variablen, die von A bis Z benannt sind, was eine Verwendung sprechender Variablennamen verhindert. Als einziger Datentyp wird 16 Bit-Integer verwendet. Das Arbeiten mit Fließkommazahlen wird nicht unterstützt, Strings werden nur als Literale zur direkten Ausgabe benutzt.

Wie in vielen BASIC-Dialekten ist das Verwenden von Unterprogrammen möglich. Der Aufruf erfolgt dabei nach Zeilennummer, das Übergeben von Parametern ist nicht möglich. Da ChipBASIC keine lokalen Variablen unterstützt, ist Rekursion nur bedingt sinnvoll.

Andererseits verfügt ChipBASIC über interessante eingebaute Funktionen für die Ausgabe von Grafik und Sound, was das Entwickeln von kleinen Spielen vereinfacht. Außerdem benötigt ChipBASIC weniger Ressourcen und läuft daher schon auf kleineren, kostengünstigeren ATmega-Varianten.

7.2. Zukunft des Projekts

Femto wird als Open Source-Projekt unter der vereinfachten BSD-Lizenz weiterentwickelt und wird beim Projektverwaltungsdienst Launchpad gehostet [1]. Auf diese Weise können interessierte Entwickler Femto für ihre eigenen Projekte nutzen und sich an der Weiterentwicklung des Interpreters beteiligen.

Eine Möglichkeit ist dabei das Portieren auf neue Plattformen, und die Entwicklung von Programmierumgebungen für diese. Eine weitere Arbeitsrichtung ist das Erweitern von Femtoscript sowohl um weitere eingebaute Funktionen, als auch um neue Sprachkonstrukte, die die Ausdrucksstärke erhöhen könnten. Beispielsweise könnte man durch Einführung eines neuen Datentyps für Funktionsreferenzen Funktionen höherer Ordnung realisieren.

Als weiterführende Aufgabe kann auch das Schreiben eines Lernhandbuchs genannt werden, mit dessen Hilfe Programmieranfänger den Einchipcomputer tatsächlich als Werkzeug zum Erlernen erster Programmierkenntnisse verwenden können.

A. Tastaturcodes

Das PS/2-Tastaturmodul für den Einchipcomputer gibt druckbare Zeichen in ISO 8859-15-Kodierung zurück. Die Codes für nicht druckbare Eingaben wurden teilweise an den ASCII-Steuerzeichen ausgerichtet, folgen aber ansonsten keinem bestehenden Standard. Deshalb sind die verwendeten Codes mit den entsprechenden Tasten hier zusammengefasst.

Code	Bedeutung	Code	Bedeutung
8	Rücklöschtaste	144	F1
9	Tabulator	145	F2
10	Eingabetaste	146	F3
27	Escape	147	F4
127	Entfernen (Del)	148	F5
128	Pfeil nach oben	149	F6
129	Pfeil nach unten	150	F7
130	Pfeil nach links	151	F8
131	Pfeil nach rechts	152	F9
132	Pos1 (Home)	153	F10
133	Ende	154	F11
134	Einfügen (Ins)	155	F12
135	Bild auf		
136	Bild ab		
137	Druck (PrintScreen)		

Literatur

[1] Die Femto-Projektseite auf Launchpad. URL http://www.launchpad.net/femto.

[2] ATmega644 preliminary datasheet, Aug 2007. URL http://www.atmel.com/dyn/resources/prod_documents/doc2593.pdf.

[3] Seminar "Machbarkeitsstudie Einchipcomputer", 2007. URL http://userpages.uni-koblenz.de/~physik/informatik/ECC/.

[4] Grégory Catellani. Die PS/2-Schnittstelle, Mai 2007. URL http://userpages.uni-koblenz.de/~physik/informatik/ECC/ps2.pdf.

[5] Jörg Wolfram. AVR-ChipBasic2: Ein BASIC-programmierbarer Einchip-Computer mit dem ATMega644, 2006-2009. URL http://www.jcwolfram.de/projekte/avr/chipbasic2/main.php.