



Diese Datei bezieht sich auf die VM der Buildversion 0.52 und ist damit für alle anderen Versionen nur beschränkt gültig.

Hier ist die Hilfe- und Referenzdatei zu meiner VM, dem zugehörigen Assembler und dem C-Compiler. Außerdem ist ein kleines Tutorial inkludiert, allerdings fängt es nicht ganz am Anfang an. Programmierkenntnisse in irgendeiner (höheren, heute verwendeten) Sprache sind für das Tutorial notwendig.

Die VM

Hier ein paar Fakten zur VM:

- Die VM emuliert keinen Standard-PC.
- Der P-Code, der auch durch einen Assembler erzeugt werden kann, ist sehr kompakt und kann durch Drag&Drop auf die VM ausgeführt werden.
- Die Entwicklung von Software für die VM erfordert Programmierkenntnisse (entweder C/C++ oder Assembler)
- Die VM ist in BlitzMax programmiert und kann daher theoretisch auch für Linux kompiliert werden und ich werde versuchen, das auch zu tun.

Der P-Code

Der P-Code ist wie gesagt sehr kompakt. Ein Hallo-Welt-Programm(ein einfaches Programm, dass in der Eingabeaufforderung den Text „Hallo Welt!“ ausgibt) ist in Assembler etwa 25 Bytes groß. Das Programm, programmiert mit P-Code meiner VM ist etwa 26 Bytes groß.

Jeder OP-Code besteht aus einem Befehlsbyte und, je nach Befehl, 0 bis 10 Parameterbytes.

Da allerdings das direkte Programmieren in P-Code sehr kompliziert ist, werde ich das nicht erklären.

Der Assemblercode

Assemblercode ist nicht wirklich mit Hochsprache vergleichbar. Alle Codezeilen, außer Labeldeklarationen, beginnen mit einem Befehl. Außerdem gibt es keine Variablen oder Funktionen. In Assembler gibt es stattdessen Labels, die diese repräsentieren. Ein Label ist eine Art Platzhalter für eine Adresse im Code. Um die Sprache benutzerfreundlich zu halten, darf der Programmierer selbst entscheiden welche Namen er den Labels gibt. Dieses Label kann dann wie eine Funktion aufgerufen werden oder als Variable verwendet werden. Für beide Varianten der Verwendung gibt es eigene Regeln und Befehle.

Die Deklaration eines Labels ist sehr einfach. Man schreibt den gewünschten Labelnamen mit einem anschließenden Doppelpunkt:

gewuenschter_label_name:

Wenn man dieses Label als Variable(Ganzzahl) verwenden möchte, muss direkt nach der Labeldeklaration der Speicherbefehl DD aufgerufen werden, der als Parameter den Startwert der Variable übernimmt:

```
variablen_label:  
DD 24
```

variablen_label wurde damit als Variable deklariert und kann nicht mehr wie eine Funktion behandelt werden.

Wenn man stattdessen lieber eine Funktion haben möchte, muss man keinen besonderen Befehl nach der Deklaration des Labels anwenden. Aber man muss den Rücksprungbefehl am Ende der Funktion platzieren:

```
funktions_label:  
; Hier Funktionsinhalt  
RET ;Rücksprungbefehl
```

Ich habe in diesen Code ein bisschen Text eingefügt. Dies funktioniert durch das Parameterzeichen Strichpunkt. Wenn man einen Strichpunkt schreibt, kann man danach jeden beliebigen Text schreiben, weil alles nach einem Strichpunkt vom Assembler ignoriert wird.

Befehlsreferenz

Label_1 und *Label_2* repräsentieren 2 verschiedene Labels, die vor der Verwendung auch definiert wurden. Wenn es sich um Variablen handelt/handeln muss, so ist das in der Beschreibung vorgemerkt.

STRING repräsentiert einen Text, der ohne Anführungszeichen geschrieben wird. VAL und PTR sind besondere Texte, die auch so in der Referenz erwähnt werden.

„*STRING*“ repräsentiert einen Text, der von Anführungszeichen eingeschlossen sein muss.

1234 repräsentiert eine x-beliebige Zahl. Diese darf aber NUR eine Zahl sein und kein Label.

Wenn es für einen Parameter mehrere Möglichkeiten gibt, werden diese durch ein '/' (Slash) getrennt.

Wenn es die Möglichkeit gibt einen Parameter auszulassen, so ist er von eckigen Klammern eingeschlossen und mit ihm das vorangehende Komma.

Wenn von einer „internen“ Funktion/Variable/... gesprochen wird, so ist mit intern gemeint, dass sich diese Ressource im Quelltext der VM befindet.

Befehl	Verwendung	Beschreibung
--------	------------	--------------

MOV	MOV Label_1,Label_2	Für Variablen. Kopiere den Wert aus Label_2 nach Label_1.
ADD	ADD Label_1 [,Label_2]	Für Variablen. Addiere den Wert aus Label_2 zu dem aus Label_1 und speichere das Ergebnis in Label_1. Wenn Label_2 nicht gegeben ist, wird 1 addiert.
SUB	SUB Label_1 [,Label_2]	Für Variablen. Subtrahiere den Wert aus Label_2 von dem aus Label_1 und speichere das Ergebnis in Label_1. Wenn Label_2 nicht gegeben ist, wird 1 subtrahiert.
MUL	MUL Label_1,Label_2	Für Variablen. Multipliziere den Wert aus Label_2 mit dem aus Label_1 und speichere das Ergebnis in Label_1.
DIV	DIV Label_1,Label_2	Für Variablen. Dividiere den Wert aus Label_2 durch den aus Label_1 und speichere das Ergebnis in Label_1.
PUSH	PUSH VAL/PTR Label_1 Wert	Hierbei wird ein Wert auf den Stack gepusht. Der Parameter VAL oder PTR darf nur bei Labels gesetzt werden. Bei VAL wird dann der Wert aus dem Label gepusht und bei PTR wird die Adresse gepusht.
POP	POP Label_1	Holt den obersten Wert vom Stack
AND	AND Label_1,Label_2	Nur Variablen. Wendet die logische Operation AND auf die zwei Label an und speichert das Ergebnis in Label_1.
OR	OR Label_1,Label_2	Nur Variablen. Wendet die logische Operation OR auf die zwei Label an und speichert das Ergebnis in Label_1.
XOR	XOR Label_1,Label_2	Nur Variablen. Wendet die logische Operation XOR auf die zwei Label an und speichert das Ergebnis in Label_1.
NEG	NEG Label_1	Nur Variablen. Macht den Wert in Label_1 negativ.
CALL	CALL Label_1	Nur Funktionen. Ruft die Funktion auf, die von Label_1

		repräsentiert wird und speichert die Rücksprungadresse auf einem internen Stack.
RET	RET	Nimmt die letzte Rücksprungadresse vom internen Stack und springt zu ihr. Falls keine auf dem Stack liegt, beendet sich das Programm ohne Fehlermeldung. (Feature, not a bug)
INT	INT STRING/1234	Ruft eine VM-interne, unabhängige Funktion auf. Mehr dazu im Kapitel „Interrupts“
CMD	CMD STRING/1234	Ruft eine VM-interne Funktion auf die mit der Kommandozeile arbeitet. Mehr dazu im Kapitel „Input/Output“
GFX	GFX STRING/1234	Ruft eine VM-interne Funktion auf die mit der Grafikeinheit arbeitet. Mehr dazu im Kapitel „Grafik“
NOP	NOP	Tut gar nichts. War notwendig, als es noch keinen Assembler gab. Hat jetzt nur noch eine nachrangige Bedeutung.
JMP	JMP Label_1	Nur Sprungmarken. Springt bedingungslos zu Label_1.
CMP	CMP Label_1,Label_2	Nur Variablen. Vergleicht die Werte in Label_1 und Label_2 und speichert das Ergebnis in einer internen Variable.
JE	JE Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass beide Werte gleich waren.
JZ	JZ Label_1	JZ tut dasselbe wie JE
JNE	JNE Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass die Werte ungleich waren.
JNZ	JNZ Label_1	JNZ tut dasselbe wie JNE
JA	JA Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass der erste Wert

		größer war, als der zweite Wert.
JB	JB Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass der zweite Wert größer war, als der erste Wert.
JAE	JAE Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass der erste Wert größer war, als der zweite Wert oder dass beide gleich waren
JBE	JBE Label_1	Nur Sprungmarken. Springt zu Label_1, wenn der letzte Vergleich mit CMP ergeben hat, dass der zweite Wert größer war, als der erste Wert oder dass beide gleich waren.
XCHG	XCHG Label_1,Label_2	Nur Variablen. Tauscht die Werte von Label_1 und Label_2 aus.
DUPL	DUPL	Nimmt den letzten Wert vom Stack und legt in zweimal wieder auf ihn.

Prinzipien der VM

Die VM emuliert einen Computertyp, der in Wirklichkeit nicht existiert bzw noch nie gebaut wurde. Es hätte auch wenig Sinn einen solchen zu bauen, da wir doch den IBM-kompatiblen PC so sehr lieben und nicht in die DOS-Zeit zurückkatapultiert werden wollen.

Die VM hat einen Arbeitsspeicher. Dieser entspricht der Dateigröße der P-Code-Datei. Die P-Code Datei wird beim Start in den Arbeitsspeicher der VM geladen.

Dazu gibt es 2 Stacksysteme: Den Userstack(PUSH / POP) und den internen Stack(CALL / RET).

Einen Stack kann man sich am Besten als Papierstapel vorstellen. Auf diesen darf man oben Papier drauflegen oder Papier herunternehmen, aber nirgendwo anders. Man darf also nur oben auf den Stack zugreifen.

Der Userstack hat uA den Sinn Werte zwischenspeichern zu können, während er in der Hochsprache seine Bedeutung für den Programmierer verliert, da er in der Hochsprache meist nur vom Compiler beeinflusst wird.

Der interne Stack kann nicht direkt vom Programmierer beeinflusst werden(das hat gewisse Gründe). Es werden die Rücksprungadressen auf ihm gespeichert. Jedes Mal, wenn man eine Funktion aufruft,

muss sich jemand merken, wo man nach Beendigung hinspringen muss. Das tut der interne Stack. Es wäre fatal, wenn der Programmierer durch schlechte Programmierung die Rücksprungadressen manipuliert. Das würde einen GAU im Bereich der VM verursachen, da sich so ein Fehler nicht zurückverfolgen lässt.

Der interne Stack verändert seine Größe dynamisch und passt sie dem Programmfluss an, allerdings gibt es beim Userstack eine Einschränkung. Dessen Größe ist nämlich statisch und auf 4 Kilobytes beschränkt. Sollte allerdings fürs erste reichen(das ist Platz für 1024 Integerwerte(!)).

interne Funktionen

Die VM stellt interne Funktionen für den Programmierer bereit, zB Input/Output - Funktionen, Grafikfunktionen, ...

Diese sind für die Programmierung mit der VM notwendig und wurden auf drei verschiedene Befehle verteilt: INT, CMD, GFX.

INT behandelt allgemeinere Funktionen, die sich keiner genauen Klasse zuordnen lassen.

CMD behandelt Funktionen die auf die Kommandozeile(Eingabeaufforderung, Terminal) zugreifen.

GFX behandelt Grafikfunktionen.

Diese Funktionen beziehen ihre Parameter, wenn notwendig, über den Userstack.

Der Aufruf einer Funktion funktioniert nun wie folgt:

Man schreibt den Befehl zu dessen Kategorie die Funktion gehört und danach den zugehörigen Befehlsnamen. Theoretisch darf man auch einen Zahlenwert schreiben, allerdings wurde jedem Zahlenwert schon ein Name zugeordnet, der das Merken vereinfachen soll.

In den folgenden Kapiteln wird es zu jedem dieser Befehle eine kurze Erklärung und eine Funktionsreferenz geben.

Falls eine Funktion einen Wert zurückgibt, so liegt dieser, nach Rücksprung aus der Funktion, auf dem Userstack.

Interrupts - interne Funktionen mit INT

Name der Funktion	Verwendung	Beschreibung
MSEC	PUSH PTR Label_1 INT MSEC	Nur Variablen. Label_1 enthält nach Aufruf die Zeit in Millisekunden, die seit dem Booten vergangen ist.
DELAY	PUSH 1234/Label_1 INT DELAY	Nur Variablen. Die Funktion wartet die Zeit in Millisekunden, die ihr übergeben wurde.

Input/Output - interne Funktionen mit CMD

Name der Funktion	Verwendung	Beschreibung
PRINT	PUSH PTR Label_1 CMD PRINT	Nur Strings. Gibt den String an der Adresse Label_1 aus und anschließend einen Zeilenumbruch.
WRITE	PUSH PTR Label_1 CMD WRITE	Nur Strings. Gibt den String an der Adresse Label_1 aus.
PUTINT	PUSH PTR Label_1 CMD PUTINT	Nur Variablen. Gibt die Ganzzahl an der Adresse Label_1 aus.
PUTVINT	PUSH Label_1/1234 CMD PUTVINT	Nur Variablen. Nimmt die letzte Zahl vom Stack und gibt sie aus.
INPUT	PUSH PTR Label_1 PUSH Label_2/1234 CMD INPUT	Nur Strings. Lässt den Benutzer einen String eingeben, der an der Adresse Label_1 gespeichert wird. Falls er länger ist als die danach auf den Stack geschobene Maximallänge, wird er gekürzt.
INPUTINT	PUSH PTR Label_1 CMD INPUT	Nur Variablen. Lässt den Benutzer eine Zahl eingeben. Diese wird in Label_1 gespeichert.
BR	CMD BR	Gibt einen Zeilenumbruch aus.

Grafik - interne Funktionen mit GFX

Grafikfunktionen ermöglichen das Öffnen eines Grafikfensters und das Bearbeiten von Selbigem durch Grundoperationen: zB Zeichnen eines Rechtecks. Für jede dieser Operationen wird eine Funktion bereitgestellt die durch GFX oder, in der Hochsprache, durch die graphics-lib angesprochen werden kann.

Die Grafikausgabe erfolgt auf Windows über einen DirectX-Treiber und auf Linux über einen OpenGL-Treiber.

Die Grafikausgabe basiert auf Double-Buffering, dh man zeichnet zuerst alles auf eine (für den Benutzer) unsichtbare Leinwand. Wenn man fertig ist, zeichnet man diese Leinwand auf den Bildschirm. Dabei ist der Bildschirm(das was man sieht) der sogenannte Frontbuffer und die Leinwand der sogenannte Backbuffer.

Es gibt wie bei MS Paint immer eine Vordergrund und eine Hintergrundfarbe. Wenn man etwas zeichnet - dann mit der Vordergrundfarbe. Wenn man den Bildschirm löscht, wird er mit der Hintergrundfarbe gefüllt.

Die meisten Grafikfunktionen übernehmen zwei oder mehr Parameter. Diese werden nur als (1p), (2p), (3p) und (4p) angesprochen.

Name der Funktion	Verwendung	Beschreibung
START	PUSH 1234/Label_1 PUSH 1234/Label_2 GFX START	Erstellt ein Grafikfenster mit der Auflösung (1p)*(2p). Achtung! Es darf nur ein Grafikfenster erstellt werden.
END	GFX END	Schließt das offene Grafikfenster wieder.
CLS	GFX CLS	Löscht den Inhalt des Grafikfensters.
FLIP	GFX FLIP	Tauscht den Frontbuffer durch den Backbuffer aus.
SETCOLOR	PUSH 1234/Label_1 PUSH 1234/Label_2 PUSH 1234/Label_3 GFX SETCOLOR	Setzt die Vordergrundfarbe. Rot=(1p) Wertebereich: 0...255 Grün=(2p) -----"----- Blau=(3p) -----"-----
SETCLSCOLOR	PUSH 1234/Label_1 PUSH 1234/Label_2 PUSH 1234/Label_3 GFX SETCLSCOLOR	Setzt die Hintergrundfarbe. Rot=(1p) Wertebereich: 0...255 Grün=(2p) -----"----- Blau=(3p) -----"-----
ALPHA	PUSH 1234/Label_1 GFX ALPHA	Setzt den Alphawert (Durchsichtigkeitsfaktor) auf (1p). Wertebereich: 0...255
ROTATE	PUSH 1234/Label_1 GFX ROTATE	Setzt den Rotationswert auf (1p) Wertebereich: 0...359 Jeder ab da gezeichnete Körper, wird zuerst um diesen Wert in Grad gedreht.
SCALE	PUSH 1234/Label_1 GFX ROTATE	Setzt den Skalierwert auf (1p). Wertebereich: 0... nicht def. Die Größe jedes von da an gezeichneten Körpers wird zuerst mit diesem Wert multipliziert. (aber nicht gespeichert)
PLOT	PUSH 1234/Label_1 PUSH 1234/Label_2 GFX PLOT	Zeichnet einen Punkt an den Koordinaten ((1p) (2p)).
LINE	PUSH 1234/Label_1 PUSH 1234/Label_2 PUSH 1234/Label_3 PUSH 1234/Label_4 GFX LINE	Zeichnet eine Linie von der Koordinate ((1p) (2p)) zu ((3p) (4p)).
RECT	PUSH 1234/Label_1 PUSH 1234/Label_2	Zeichnet ein Rechteck. Das linke , obere Eck an der Koordinate

	PUSH 1234/Label_3 PUSH 1234/Label_4 GFX RECT	((1p) (2p)) und das rechte, untere Eck an der Koordinate ((3p) (4p)).
OVAL	PUSH 1234/Label_1 PUSH 1234/Label_2 PUSH 1234/Label_3 PUSH 1234/Label_4 GFX OVAL	Zeichnet ein Oval, das durch das umliegende Rechteck definiert wird. Das linke , obere Eck des Rechtecks ist an der Koordinate ((1p) (2p)) und das rechte, untere Eck an der Koordinate ((3p) (4p)).
TEXT	PUSH PTR Label_1 PUSH 1234/Label_2 PUSH 1234/Label_3 GFX TEXT	(1p) muss eine Stringvariable sein. Der String an Label_1 wird an der Koordinate ((2p) (3p)) ausgegeben(im Grafikfenster).

Kommentare

Im Assembler gibt es nur einzeilige Kommentare die durch einen Strichpunkt eingeleitet werden.

```
PUSH PTR Hallo
CMD PRINT ; Hier wird ein String ausgegeben
END

Hallo:
DS „Hallo Welt!“,10,0
```

In der Hochsprache lassen sich sowohl einzeilige als auch mehrzeilige Kommentare erstellen, wobei die Einleitung (und Ausleitung) der von C/C++ entspricht:

```
/*
    Dies ist ein etwas
    sinnloser mehrzeiliger
    Kommentar, der zeigen
    soll, dass auch mehrzeilige
    Kommentare funktionieren.
*/
void main(){
    print(„Hallo Welt!“); //Hier wird ein String ausgegeben
}
```

Kommentare sind unbedingt notwendig wenn:

- man in Assembler programmiert (und man Hilfe braucht oder der Code lang ist).
- man in der Hochsprache programmiert und der Code lang ist.

Kleine Einführung in die Hochsprache

Hier nun eine kleine Einführung in die von mir definierten Hochsprache.

Zuerst ein paar Fakten. Meine Hochsprache...

- hat eine C/C++-Ähnliche Syntax.
- Unterstützt die meisten Konstrukte imperativer Programmiersprachen.
- Wird in meine Assemblersprache übersetzt.

Das unterstützt die Sprache nicht:

- Arrays von Integern(*)
- Strukturen(*)
- Klassen
- Symbolische Konstanten(*)
- Unions
- Zeiger im herkömmlichen Sinn
- Lokale Variablen

Alle mit einem (*) gezeichneten Einträge werden hoffentlich bald implementiert.

Das Programm beginnt mit der main-Funktion, die vom Programmierer definiert werden MUSS. Diese Funktion liefert keinen Wert zurück und ist daher vom Typ void.

Besonderheit: Wenn man eine Funktion definieren will, die vom Typ void ist, so kann man das Schlüsselwort void weglassen.

Zulässig sind daher:

```
void main() {  
}
```

Und

```
main() {  
}
```

Eine leere Klammer bei Funktionsdeklarationen ist nach ANSI-C eine Funktion, die unendlich viele Parameter übernehmen kann. Mein Compiler handhabt solche Funktionen allerdings, wie die ersten C-Compiler, als Funktionen ohne Parameter.

Auch die Vorwärtsdeklaration von Funktionen ist möglich - so wie in C/C++.

Das Termrechensystem kann nicht mit Termen der Infix-Notation umgehen. Es kann nur die Postfix-Notation(UPN) parsen.

Eine Funktion, der Parameter übergeben werden leitet diese Parameterwerte in globale Variablen um. Eine Deklaration sieht zB so aus:

```
int x=0;

int double(x){
    x=x 2 *;
    return x;
}

main(){
    print(double(5)); //Am Bildschirm erscheint 10
}
```

=> Der Parameter wird in die Variable x umgeleitet und diese dann manipuliert und zurückgegeben.

#include existiert auch in meiner Hochsprache und funktioniert wie in C/C++. Mit der Direktive lassen sich Quelltexte aus anderen Dateien inkludieren. Dies ist nützlich, wenn man zB die graphics-lib verwendet.

Static Variablen sind Variablen, die nur in der Datei angesprochen werden können, in der sie deklariert sind. Dies setzt allerdings voraus, dass die Dateien, die zusammengefügt werden, unterschiedliche Namen haben. Diese Variablen werden durch das Schlüsselwort *static* vor *int* deklariert.

Konstrukte die eine Bedingung erfordern (dh if, while, for, etc.) können nur eine dieser verarbeiten. Man kann NICHT mehrere Bedingungen durch && oder || verbinden. Es ist bis jetzt nur eine Bedingung erlaubt:

```
static int x=5,y=7;

void main(){
    if(x==y){                // zulässig
        print(„5 == 7“);
    }
    if(x==y && 2==3){        // unzulässig
        print(„5 == 7 und 2 == 3“);
    }
}
```

Auch Terme sind nur bei Zuweisungen erlaubt:

```
int x=5;
void main(){
    x=25 5 / 8 +; //zulässig
    if(x==25 5 / 8 +){ //unzulässig
    }
}
```

Kleines Tutorial

So, ich werde mir nun Mühe geben das ganze Schritt für Schritt zu erklären. Ich möchte aber nochmal darauf hinweisen, dass dies ein Tutorial für Leute ist, die schon Programmiererfahrungen gemacht haben.

Als erstes möchte ich euch die Main-Funktion vorstellen, denn mit ihr startet jedes Programm und ohne sie, ist ein Programm nicht ausführbar. Außerdem darf es sie nur einmal geben.

```
void main(){
}
```

Das ist das erste Programm. Wie führt man es nun aus?

Man speichert die Datei, die man ausführen will in dem Ordner projects, der mit der VM runtergeladen wurde. Die Datei muss aber die Endung .c haben. Dann öffnet man die Eingabeaufforderung/das Terminal und bewegt sich mit cd in das Verzeichnis der VM. Dann tippt man folgendes ein:

```
vm projects\dateiname.c
```

In dem Moment wird die Datei von einem Präprozessor abgearbeitet, dann nach Assembler kompiliert, vom Assembler assembliert und schließlich von der VM ausgeführt, allerdings so schnell, dass man fast keinen Zeitunterschied bemerkt.

Immer wenn man in der Datei etwas ändert, muss man sie so aufrufen, wie oben erklärt. Doch wenn man die Datei schon einmal so aufgerufen hat und danach nichts geändert hat, darf man die Endung .c weglassen, dann wird nur die VM aufgerufen - dh es wird nicht präprozessiert, kompiliert oder assembliert(!).

Doch, dieses Programm tut nichts. Nein - Es ist nur die Grundstruktur, die man nicht vergessen sollte.

Innerhalb der Funktion main() die durch { und } begrenzt ist, kann man nun den ausführbaren Code schreiben.

```
void main(){
    print(„Hallo Welt!“);
}
```

Wenn man dieses Programm ausführt, sollte der Text „Hallo Welt!“ (ohne Anführungszeichen) im Konsolenfenster erscheinen. Print ist der in den Compiler integrierte Ausgabebefehl, der dem Benutzer viele Freiheiten lässt. Folgender Code führt auch zum selben Ergebnis:

```
void main(){
    print(“Hal”+“lo”+“ W”+“elt”+“!”);
}
```

Das heißt man kann Strings, die durch Anführungszeichen eingeschlossen sind durch ein + zusammenhängend ausgeben.

Nun zu Variablen. Meine VM/mein Assembler/mein Compiler kennt nur zwei Datentypen, nämlich String(statisch) und Int. Zuerst werde ich etwas zu Integervariablen erklären:

```
int x,y;
void main(){
    print(x+“ : ”+y);
}
```

x und y sind nun Variablen vom Typ int, dh sie sind Ganzzahlen. Da ich sie nicht explizit initialisiert habe – also, ihnen keinen Wert zugeordnet habe – bekommen sie vom Compiler den Wert 0 zugewiesen. Am Bildschirm erscheint „0 : 0“ (ohne Anführungszeichen).

Explizite Initialisierung funktioniert so:

```
int x=25,y=16777216;
void main(){
    print(x+“ : ”+y);
}
```

Am Bildschirm erscheint nun „25 : 16777216“ (ohne Anführungszeichen).

Doch auch während dem Programmverlauf lassen sich die Werte ändern. Die Zuweisung unterstützt keine Infix-Notation – die Darstellung von Termen, die man normalerweise in der Schule lernt. Dh Terme die so aussehen:

$x=25*((7+2)-23*7)$

sind unzulässig. Man muss diesen Term zuvor in die UPN umformen:

$x=25\ 7\ 2\ +\ 23\ 7\ *\ -\ *$

Ich werde hier nicht auf die UPN eingehen, da das ein Tutorial zu meiner VM ist. Auf Wikipedia wird sie allerdings sehr gut erklärt und ich möchte auf diese Enzyklopädie verweisen.

```
int x=25,y=16777216;
void main(){
    x=25 7 2 + 23 7 * - *;
    print(x+" : "+y);
}
```

Nun bekommt x einen neuen Wert zugewiesen, bevor es ausgegeben wird.

If-Abfragen funktionieren folgendermaßen:

```
int x=25,y=16777216;
void main(){
    if(x==y){
        print(„Das ist falsch und wird nicht ausgegeben“);
    }else{
        print(„Das ist richtig und wird ausgegeben“);
    }
}
```

Das else kann man weglassen. In C/C++ gibt es auch die Möglichkeit mehrere else if Zweige einzubauen, dies ist auch in meiner Hochsprache möglich.

Es ist auch in if, while, for, etc nur eine Bedingung erlaubt. Verbindungsstücke wie && oder || wurden nur teilweise implementiert. Daher kann ich die Verwendung nicht empfehlen.

```
int x=25;
void main(){
    while(x>=0){
        print(x);
        x--; // Zieht von der Variable x 1 (eins) ab.
    }
}
```

So sieht die While-Schleife in meiner Hochsprache aus - eigentlich fast wie in C/C++. Die Vergleichsoperatoren sind folgende: ==, !=, >, <, >=, <=

Falsch ist: ==>, ==<, =

Auch do-while gibt es:

```
int x=25;
void main(){
    do{
        print(x);
        x--;
    }while(x>=25);
}
```


Aber am schönsten ist eine Zählschleife mit for:

```
int x;
void main(){
    for(x=25;x>=0;x--){
        print(x);
    }
}
```

Funktionen lassen sich exakt so definieren wie main(). Wenn man einen Wert zurückgeben möchte, muss man allerdings statt dem void ein int schreiben:

```
static int zx;

int doubler(zx){
    zx=zx 2 *;
    return zx;
}

void Einleitung(); /* Dies ist eine Vorwärtsdeklaration. Durch sie
                    kann man Funktionen, die eigentlich erst
                    nach deren Aufruf definiert wurden
                    deklarieren.
                    In dem Fall wäre das nicht notwendig, aller-
                    dings ist es für den rekursiven Einsatz
                    wichtig!
*/

int x=3;
void main(){
    Einleitung();
    print(x); // Gibt 3 aus
    x=doubler(x);
    print(x); // Gibt 6 aus
}

void Einleitung(){
    print(„Willkommen!“);
}
```

Die Eigenheiten meiner Hochsprache bei Funktionen sind sehr außergewöhnlich, daher möchte ich dazu raten auch den Teil „kleine Einführung in die Hochsprache“ nach diesem Tutorial zu lesen.

Schlusswort:

Nun. Ich hoffe, dieses Dokument ist hilfreich und halbwegs gut geschrieben. Ich weiß meine VM, mein Assembler und mein C-Compiler werden keine große Verbreitung finden, doch es ist das erste größere Projekt, das ich programmiert habe. Ich will es unbedingt bis zu einem Release drängen.

Fragen/Wünsche/Anregungen/Beschwerden: entweder per PN im BlitzForum(falls sie schon registriert sind) oder per E-Mail an programmer7@hotmail.com

Auch Bugs oder Fehler können Sie mir an die Adresse mailen. Am besten mit genauer Beschreibung (eventuell Screenshot) und allen Codedateien. Davor sollten Sie allerdings prüfen, ob sie die aktuelle Version verwenden.

© by Christian Fiedler