

Entstehungsgeschichte

Ursprünglich war die Beteiligung an dem AVR-Basic Projekt von Uwe Berger gar nicht so wirklich geplant. Das ganze ist dann aber irgendwie zum Selbstläufer geworden, zumal ich ebenfalls die Jahre über immer mal wieder an einem eigenen Basic was programmiere, bzw Ansätze ausprobiere. Ich war überrascht mit welch einfachen Mitteln man ein einigermaßen brauchbares Basic zusammenbauen kann. Und so hab ich mich da mal ein bischen „ausgetobt“.

Herausgekommen ist wie ich finde ein recht brauchbarer Ableger der Ursprungsversion von Adam Dunkels, der einige interessante Features bietet, und mit dem Compiler auch sehr (!) schnell ist.

Warum zum Kuckuck Basic auf einem AVR laufen lassen ?

Zugegeben ist der Einsatz des Basics auf einem AVR nicht gerade eine „zeitsparende“ Angelegenheit. Aber zum einen wird das durch den Einsatz des Compilers wieder mehr als wett gemacht (man kann schon fast von einem JIT-Compiler reden) wird.

Zum anderen ist das Version nicht nur für einen AVR gedacht, sondern für (nahezu) jedwede Plattform. Durch ein wenig „Tricksen“ (so schlimm ist es nicht) lassen sich ein paar Eigenheiten des AVR's auf anderen Plattformen transparent programmieren. Man programmiert Quasi also „für“ den AVR. Beim Einsatz auf anderen Plattformen werden die Eigenheiten (z.b. `pgm_read_byte`) durch Makros aufgelöst, wodurch sich ein dennoch transparentes Verhalten ergibt (meistens jedenfalls :-))

Dennoch ist der Einsatz eines Basics auf dem AVR durchaus sinnvoll. Z.b. Einfache Steuerungen die flexibel einsetzbar sein sollen. Oder bei entsprechender Erweiterung einfache GUIs. Aber auch ein einfaches Entwicklungssystem sind durchaus denkbar und relativ einfach zu implementieren.

Funktionsweise

Der Ursprungsinterpreter von Adam Dunkels funktioniert sehr einfach. Es gibt einen Tokenizer, der einfach nur Text (und dazugehörige Informationen) in Tokens umsetzt. Eine Zahl bekommt nun ein Token und die Information des Wertes (getrennt abgelegt). Die erkannten Tokens werden dem eigentlichen Basic-Interpreter zugeführt. Der erwartet bestimmte Tokens (z.b. Zeilennummer) und geht dann Schritt für Schritt die erwartete Tokenkette durch, verzweigt ggf. in Statements oder in rekursiven Aufrufen wie bei einem Ausdruck. Die einzigen Möglichkeiten sich „bemerkbar“ zu machen sind in der Ursprungsversion ein PRINT. Dadurch das der Code recht einfach gehalten ist lässt sich das Basic sehr einfach erweitern.

Meine Version beinhaltet folgende Erweiterungen :

- Compiler-Unterstützung
- Zeichengerät
- C-Erweiterungen
- Funktionen als Operatoren
- Tokentyp sowie „Zeigertyp“ definierbar
- Fehlertexte aktivieren/deaktivieren
- Weitere Basic-Statements können „eingehängt“ werden
- weitestgehende Kapselung

Anbindung

Die Anbindung des uBasics-0.5 ist relativ einfach. Man übergibt den Text an den Tokenizer. Die Übergabe funktioniert im einfachsten Fall über einen Zeiger auf den Text im RAM. Um das Basic-Programm auszuführen, führt man in einer Schleife `ubasic_run ()` aus, die dann beendet wird wenn das Ergebnis der Funktion `ubasic_finished()` ungleich 0 ist.

Da das RAM auf einem AVR begrenzt ist gibt es die Möglichkeit ein Zeichengerät zu verwenden. Dieses kapselt den Zugriff auf den Text, womit es möglich wird unterschiedlichste Medien (Flashspeicher, SD-Karte, div. Blockmedien, externes SRAM, usw.) zur Ausführung des Basic-Codes (in Textform ebenso wie in vorcompilierter Form) verwenden zu können. Die Zeichengeräte bestehen aus lediglich 5 einfachen Funktionen, die irgendwo im restlichen Quellcode vorhanden sein müssen. Die externals dazu werden automatisch erzeugt.

Bei C-Erweiterungen sind die Funktionen und Variablen ebenfalls automatisch als externals eingebunden und müssen sich so lediglich irgendwo im restlichen Quellcode befinden. Die Erweiterungen umfassen den Aufruf von C-Funktionen, den Zugriff auf C-Integer-Variablen (lesend wie schreibend), sowie den Zugriff auf C-Arrays (ebenfalls lesend und schreibend). Die Definition der Erweiterungen werden in der Konfigurationsdatei „`basic_cfg.h`“ im Abschnitt `BASIC_USES_EXTENSIONS` festgelegt und es können einfach weitere Einträge hinzugefügt werden, die den Sprachumfang und die Möglichkeiten erweitern.

Für die Compiler-Unterstützung benötigt man eine Funktion die ein Byte wegschreiben kann. Sinnvollerweise sollte es entweder das SRAM sein oder aber eines der verwendeten Medien wohin die erzeugten Bytes verfrachtet werden. Es gibt für den Compiler eine einzige Funktion, die den vorliegenden Basic-Quell-Text in Tokens übersetzt und diese mit den dazugehörigen Daten binär wegschreibt. Weiters werden in dieser Funktion die `Gosub/Goto` Zeilennummern in Sprungmarken relativ zum Anfang des Codes ersetzt. Dadurch erspart man sich das langwierige Suchen (bzw erneute Durchlaufen des gesamten Quelltextes) nach der Zeilennummer. Diese beiden Optimierungen ergeben insgesamt einen extrem hohen Geschwindigkeitsvorteil, bei gleichzeitiger Reduzierung der ursprünglichen Quellcode-Größe.

Um eine Bereichsprüfung bei Operatoren zu ermöglichen, kann man die Operatoren durch eigene Funktionen ersetzen. Dadurch ist es möglich z.b. bei Division durch 0 einzuschreiten. Primär ist es für einen späteren Zeitpunkt gedacht, wenn z.b. andere Datentypen mit ins Spiel kommen. Aber dann muß denke ich eh ein anderer Weg her.

Zeichengerät

Zeichengeräte können durch hinzufügen von Einträgen in der Konfigurationsdatei „`basic_cfg.h`“ im Abschnitt `TOKENIZER_MEDIUM` erweitert werden. Wie schon erwähnt bestehen Zeichengeräte aus jeweils 5 Funktionen. Die erste Funktion (`SET_TEXT`) beinhaltet das Setzen des auszuführenden bzw übersetzen des Textes/Binärcode.

Da das Zeichengerät sich transparent verhalten soll wird der Funktion ein untypisierter `void *`-Zeiger übergeben, sowie die Länge des Textes. Wie der Zeiger und die Länge interpretiert werden, hängt vom Zeichengerät selbst ab. Es wäre z.b. denkbar das sich hinter dem `void *`-Zeiger ein Zeiger auf einen Dateinamen verbirgt da das Zeichengerät Texte von der SD-Karte einliest und einen Dateinamen erwartet. Die Länge gibt dann die Länge des Programms an.

Ein anderes Beispiel wäre das das Zeichengerät ein externes SRAM anspricht. In dem Falle würde der `void *`-Zeiger auf einen 32 bit Wert Zeigen der die Adresse im SRAM angibt an der der Text beginnt. Aber auch ein „normaler“ `char *`-Zeiger könnte sich hinter dem `void *`-Zeiger verbergen wenn man einfach nur das interne SRAM als Zeichengerät verwendet.

Die zweite Funktion (GET_CHAR) liefert das aktuelle Zeichen an der internen Zeigerposition des Zeichengerätes zurück. Der Zeiger wird dabei nicht neu positioniert. Das passiert in der dritten Funktion (NEXT_CHAR). Diese setzt den Zeiger ein Zeichen weiter. Ob dabei (im Falle einer SD-Karten-Anbindung) beispielsweise eine neue Seite geladen werden muß entscheidet diese Funktion ebenfalls. Es wird dabei kein Zeichen zurückgegeben. Die letzten beiden Funktionen beinhalten ebenfalls Positionier-Operationen. Die vierte Funktion liefert den Zeiger (bezogen auf den Anfang des Textes) zurück. Die fünfte und letzte Funktion setzt diesen Zeiger (ebenfalls bezogen auf den Anfang des Textes). Falls dabei eine neue Seite geladen werden muß wird dies ebenfalls in dieser Funktion abgewickelt.

Durch Verwendung dieser 5 Funktionen können viele Quellen als Eingabemedium für das Interpretieren des Basics verwendet werden. Somit ist es (theroretisch) möglich weitere „Basic-Unterprogramme“ von externen Medien auszuführen, während das „Hauptprogramm“ nach Beendigung des Unterprogramms wieder weiter ausgeführt wird. Lediglich ein neuer Variablen-Kontext für das Unterprogramm müsste geschaffen werden.

Das Demo enthält zwei Zeichengeräte. Ein einfaches Zeichengerät das einfach nur aus dem RAM ausliest, und ein Zeichengerät das ebenfalls aus demselben RAM ausliest, aber dafür Blockoperationen benutzt, also einen eigenen kleinen Buffer enthält der dann Seitenweise mit dem RAM-Buffer gefüllt wird. Dieses Zeichengerät kann als Grundlage für die Anbindung an beispielsweise eine SD-Karte dienen.

Wird keine Zeichengeräte Unterstützung aktiviert, so werden die Makros die auf die Zeichengeräte-Funktionen gemappt sind einfach durch „normale“ Zeigeroperationen ersetzt, sodaß der Compiler dort sehr gut optimieren kann. Allerdings beziehen sich die Hole/Setze-Operationen des Zeigers ebenfalls auf den Anfang des Quelltextes, um zu der Zeichengerätefunktionalität kompatibel zu sein.

C-Erweiterung

Die C-Erweiterungen umfassen wie schon angesprochen das Aufrufen von C-Funktionen (mit Parametern), das lesen und schreiben von C-Variablen, sowie C-Arrays. Die aufzurufenden C-Funktionen haben alle (!) dieselbe Syntax :

```
I16 name (T_USER_FUNC_DATAS stParams)
```

Der Rückgabewert ist 16 Bit Signed und kann direkt in Basic-Ausdrücken verwendet werden, sprich es gibt in dem Sinne nur Funktionen und keine Prozeduren (ohne Rückgabewert).

Der Typ der übergebenen Parameter besteht aus einem Array aus Einträgen. Jeder eintrag enthält ein Byte welches den Typ widerspiegelt der den aktuellen Parameter repräsentiert. Zur Zeit gibt es nur Integer und String. Der Eintrag besteht weiterhin aus einer Union eines Integers und eines Strings. Die Prüfung auf korrekte Übergabe der Typen muß von der aufzurufenden Funktion selbst vollzogen werden. Dafür gibt es im uBasic-0.5 noch keinen Mechanismus.

C-Variablen können wie schon erwähnt beschrieben und gelesen werden.

Bei C-Arrays gilt dasselbe, allerdings lässt sich hier noch mit einem weiteren Parameter in der Definition die Array-Größe festlegen. Im uBasic wird diese Grenze respektiert und auch geprüft. Als Erweiterung ist vorgesehen das die Typ-Prüfung bei C-Aufrufen noch durchgeführt wird, sowie bei den C-Variablen/Arrays ein Modus gesetzt werden kann der Variablen/Arrays vor dem beschreiben/auslesen schützt, quasi ein Read/Write-Flag. Außerdem soll es die Möglichkeit geben, die Arrays nicht direkt zu beschreiben/zu lesen, sondern die Werte über eine Funktion zu verarbeiten. Damit lässt sich eine dynamische Größe des Arrays realisieren. Weiterhin wäre dies die Kapselung wie man sie von den Properties in der WIN32 Programmierung her kennt.

Compiler-Unterstützung

Der Compiler ist mit das interessanteste Feature am ganzen uBasic-0.5 Komplex. Mit diesem ist es Möglich die reinen Basic-Texte als binäre Tokenliste (inkl der ggf vorhandenen weiteren Informationen) inkl ersetzten Sprungmarken für Gosub/Goto-Aufrufe abzuspeichern, und aufzurufen. Die Abarbeitung des Basic-Quelltextes erfolgt im Binär wie im Text-Modus gleichermaßen, bzw unterscheidet sich nur geringfügig in einzelnen Statements. Dadurch wird der Code transparent ausgeführt. Der Geschwindigkeitsvorteil der sich ergibt ist allerdings enorm. Je nach Art des Basic-Programms ist die Abarbeitung ein vielfaches schneller und selbst der Compilierungsvorgang inkl Ausführung ist mehrfach so schnell als wenn der Text uncompiliert abgearbeitet wird.

Um das Compilerfeature nutzen zu können bedarf es einer einfachen Funktion die dem Compiler mit übergeben wird, welche ein vom Compiler erzeugtes Byte wegschreibt. Sinnigerweise ist diese Schreibfunktion so geartet das ein vorhandenes TOKENIZER_MEDIUM diesen Byte-Strom wiedergeben kann. Somit können getestete Programme in Binär-Darstellung übersetzt werden und arbeiten sehr schnell.

Einen direkten vergleich der Geschwindigkeit sieht man in dem Demo wenn man "load 4 run" gegenüber "test 4" eintippt. Im ersten Fall wird das Programm 4 in das RAM geladen und ganz normal ausgeführt. "test 4" hingegen ist die Abkürzung für "load 4 compile run compiled". Mit diesem einfachen Programm wird der Geschwindigkeitsvorteil sofort sichtbar. Noch gravierender ist das mit dem Testprogramm Nummer 7. Da wird 10x10x10 ein print ausgegeben. Die Geschwindigkeit des compilierten Programmes spricht für sich ...