Dokumentation uBasic (auf einem AVR)

Uwe Berger (bergeruw@gmx.net)

Stand: 12.04.2011

Inhalt

1 Motivation

Welcher Mikrocontroller-Programmierer kennt das Problem nicht: man hat eine schicke Firmware auf den MC gebrannt, braucht schnell eine neue (einfache) Funktionalität und will/kann nicht gleich an den C-Code der Firmware ran. Was liegt also näher, Funktionen beliebig "nachladen" und ausführen zu lassen? Dieses Ansinnen mit Binär-Code-Fragmenten zu machen, dürfte ein schwieriges, wenn nicht sogar unmögliches Unterfangen sein. Und man ist wieder von einem Compiler abhängig. Script-Sprachen sind da viel besser geeignet, da sie verständlicher und leichter zu programmieren sind. Voraussetzung ist dabei natürlich, dass ein entsprechender Script-Interpreter auf der Zielplattform verfügbar ist.

Jetzt könnte man natürlich einen eigenen und u.U. speziell für Mikrocontroller angepassten Sprachsyntax entwickeln und implementieren (Bsp. ECMDSript bei ethersex). Viel sinnvoller erscheint es aber, wenn man auf alt bewährte und bekannte Dinge zurückgreift: jeder (alte) Programmierer hat mal mit Basic angefangen oder zu mindestens davon gehört! Der Zufall wollte es, dass Adam Dunkle ein C-Gerüst für einen kleinen und ressourcenschonenden TinyBasic-Interpreter (uBasic) veröffentlicht hat. Mit minimalen Modifikationen ist das Ding auch auf einem AVR sofort lauffähig und beeindruckt durch den geringen Ressourcen-Verbrauch.

Auf dieses Gerüst aufsetzend, entstand uBasic für AVRs.

Ziel ist es, einen universellen Basic-Interpreter zu haben, der in AVR-Programme einfach eingebunden werden kann. Die Basic-Programme können über vorhandene Schnittstellen (seriell, Ethernet o.ä.) geladen werden oder sind auf einem externen Speichermedium (SD-Card, Dataflash o.ä.) verfügbar und einlesbar.

Folgende Eigenschaften weist uBasic auf:

- relativ einfach zu verstehender Quellcode, der es nach kurzer Einarbeitungszeit möglich macht, den Sprachumfang zu erweitern
- modular aufgebaut, einfach in eigene Applikationen integrierbar
- kleiner übersetzter Code, geringer Speicherverbrauch
- Basic-Syntax ist an das bekannte TinyBasic angelehnt
- Sprachumfang und interner Speicherverbrauch parametrierbar
- Basic-Befehle für den Zugriff auf Funktionen und Variablen der einbindenden Firmware vorhanden
- einfache Schnittstelle zur Implementierung eigener Zugriffsmethoden auf das Speichermedium des Basic-Quelltextes

2 Sprachumfang

2.1 Generell

Der Sprachumfang lehnt sich, mit einigen Einschränkungen (siehe weiter unten), an das bekannte TinyBasic an.

Basic-Zeilen fangen mit einer eineindeutigen Zeilennummer an, diese werden in der folgenden

Syntaxbeschreibung nicht mit aufgeführt! Innerhalb des Interpreters wird die Richtigkeit der Zeilennummer (Eineindeutigkeit, Reihenfolge) nicht abgeprüft. Sie spielen aber bei Sprungbefehlen (*goto*, *gosub-return*) sowie *for-to-next* schon eine entscheidende Rolle!

Alternativ kann, die entsprechende Konfiguration vorausgesetzt, die Zeilennummerierung weggelassen werden. Sprungziele (GOTO, GOSUB) sind natürlich weiterhin zu nummerieren.

Innerhalb von uBasic wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Ein Basic-Programm endet mit an einer *end*-Anweisung.

In der Folge werden:

- Befehle ohne einschließende Klammern ö.ä. angegeben
- erforderliche Syntaxelemente in <...> angegeben
- alternative Syntaxelemente mit einem | getrennt
- optionale Syntaxelemente in [...] angegeben

Kombinationen sind möglich.

Bedeutung innerhalb folgender Syntaxbeschreibung:

- *val* → ein numerischer Wert
- *str* → ein String (eingeschlossen in "...")
- *expr* → eine Expression (Ausdruck)
- *var* → eine Variable
- *rel* → Relation (Vergleichsoperation)
- ... → weitere beliebige Statements/Zeichen o.ä.

2.2 Einschränkungen

Folgende Einschränkungen gelten für Basic-Programme:

- systembedingte Einschränkungen:
  - Schachtelungstiefe for-Befehl: ist begrenzt und via define einstellbar (uBasic_config.h)
  - Schachtelungstiefe gosub-Befehl: ist begrenzt und via define einstellbar (uBasic_config.h)
  - Länge des Basic-Quelltext: ist begrenzt durch vorhandene Speichergrößen
- im Vergleich zu anderen verbreiteten Basic-Dialekten (außer TinyBasic):
  - Wertebereiche Variablen: nur Integer (signed)
  - Variablennamen: nur ein Buchstabe von 'a'...'z' bzw. 'A'...'Z' (es gilt a=A usw.)
  - Anzahl Variablen: max. 26 und via define einstellbar (uBasic_config.h)

2.3 uBasic-Syntax

2.3.1 Darstellungsformen für Zahlen

Es sind innerhalb eines Basic-Programmes folgende unterschiedliche Darstellungsformen für Zahlen zulässig:

- Dezimalzahlen: 1234
- Hexadezimalzahlen: 0x12AB
- Dualzahlen: 0b10101010

Die Ausgabe (PRINT-Anweisung) von Zahlen erfolgt prinzipiell dezimal.

2.3.2 Operatoren

Arithmetische Operatoren

Operator	Bedeutung
+	Addition zweier Werte; z.B. *1+2*
-	Subtraktion zweier Werte; z.B. *3-2*
/	Division zweier Werte; z.B. *4/2*
*	Multiplikation zweier Werte; z.B. *3*5*
%, mod	Modulo (Divisionsrest); z.B. *5%4* bzw. *5 mod 4*

Bit-Operatoren

Operator	Bedeutung
|,or	bitweise OR-Verknüpfung; z.B. *2|3* bzw. *2 or 3*
&, and	bitweise AND-Verknüpfung; z.B. *2&4* bzw. *2 and 4*
xor	bitweise XOR-Verknüpfung; z.B. *3 xor 6*
shr	Rechtsverschiebung; z.B. *4 shr 1*
shl	Linksverschiebung; z.B. *4 shl 1*

Vergleichsoperatoren

Operator	Bedeutung
<	kleiner
>	größer
=	gleich
>=	größer gleich
<=	kleiner gleich

<>	ungleich

Hinweis: diese Vergleichsoperatoren sind nur in *if-then-else*-Anweisungen einsetzbar, nicht in *for-next*-Schleifen (bis auf '=' natürlich...).

2.3.3 Schleife von/bis (for/to/next)

```
for var=<val|expr|var> to|downto <val|expr|var> [step <val|expr|var>]
...
next <var>
```

Schachtelungen von *for-next*-Schleifen sind möglich.

2.3.4 Bedingte Anweisung (if/then/else)

```
if <var|val|expr><rel><var|val|expr> then ... [else ...]
```

Es ist eine Kurzform der *if*-Anweisung zulässig, bei der das *then* weggelassen werden darf. Bei Verwendung dieser Kurzform darf es dann aber keinen *else*-Zweig geben. Es wird dann eine entsprechende Fehlermeldung ausgegeben und das Programm abgebrochen.

Korrektes Beispiel:

```
IF A=1 GOTO 100
```

Fehlerhaftes Beispiel:

```
IF A=1 GOTO 100 ELSE GOTO 200
```

Hinweis: Nach *then* bzw. *else* ist nur ein Statement oder Ausdruck möglich. Sollten jeweils längere Programmstücke notwendig sein, kann man sich mit Unterprogrammen (*gosub-return*) behelfen.

2.3.5 Sprung (goto)

```
goto <val>|<expr>|<var>
```

val gibt eine gültige Zeilennummer an bzw. das Ergebnis von *expr* oder *var* ist eine solche, zu der im Kontext gesprungen werden soll.

2.3.6 Unterprogramm aufrufen (gosub)

```
gosub <val>|<expr>|<var>|<str>
...
return
```

val eine gültige Zeilennummer an bzw. das Ergebnis von *expr* oder *var* ist eine solche, an der das Unterprogramm beginnt. Ende-Anweisung im Unterprogramm ist *return*, es wird zur Zeile nach dem ent-

sprechenden *gosub*-Befehl zurückgesprungen und die Abarbeitung des Programms fortgesetzt.

Mit *str* ist es möglich ein externes Unterprogramm aufzurufen. Extern heißt, der Quelltext ist nicht im aktuellen Basic-Programm enthalten, sondern befindet sich z.B. in einer eigenen Datei. Das Unterprogramm wird dabei geöffnet, von der ersten Zeile abgearbeitet und muss zwingend mit einem *return*--Befehl enden, um in das aufrufende Programm zurückzukehren. Variableninhalte usw. bleiben über die Programmgrenzen hinaus erhalten. Siehe auch entsprechendes Kapitel in dieser Dokumentation.

Schachtelungen von *gosub*-Anweisungen sind möglich, wobei aber die Schachtelungstiefe begrenzt ist (siehe Einstellungen in ubasic_config.h).

2.3.7 Programm-Ende (end)

```
end
```

Beendet ein Basic-Programm. Unterprogramme (*gosub-return*) sollten nach der Ende-Anweisung stehen.

2.3.8 Zufallswert (srand/rand)

Zufallsgenerator initialisieren:

```
srand
```

Einen Zufallswert zwischen 0 >= ... <= (Wert in Klammern) erzeugen:

```
<var>=rand(<val|var|expr>)
```

Kann auch selbst Element eines Ausdrucks sein.

Für die Implementierung der Zufallsfunktionen auf Nicht-AVR-Plattformen wurden die entsprechenden Routinen der libc verwendet. Als seed-Wert für srand() dient die aktuelle Uhrzeit.

Für AVR-Plattformen werden nicht die entsprechenden Routinen aus der libc verwendet. Der seed-Wert wird aus dem aktuellen Speicherinhalt des SRAM errechnet, als rand()-Funktion wird auf ein Algorithmus zurückgegriffen, der u.a. auf der Webseite http://www.firstpr.com.au/dsp/rand31/ zu finden ist.

2.3.9 absoluter Betrag (abs)

```
<var>=abs(<val|var|expr>)
```

Kann auch selbst Element eines Ausdrucks sein.

2.3.10 Complement (not)

```
<var>=not(<val|var|expr>)
```

Kann auch selbst Element eines Ausdrucks sein.

2.3.11 Ausgabe (print)

```
print <val|var|expr|str> [<,|;> [<val|var|expr|str>]]
```

Zwischen den einzelnen Ausdrücken wird ein Leerzeichens ausgegeben, wenn als Trennzeichen ein Komma angegeben wird. Ein Semikolon unterdrückt die Ausgabe des Leerzeichen. Stehen Komma oder Semikolon am Zeilenende, wird der Zeilenvorschub unterdrückt.

Die Ausgabe erfolgt auf der "Standard-Ausgabe", welche in den entsprechenden Defines in ubasic_config.h definiert ist.

2.3.12 Variable setzen (let)

```
[let] <var>=<var|val|expr>
```

Die Angabe von *let* ist optional. Variablen sind immer vom Typ signed Integer. Beim Start eines Basic-Programms werden sie mit 0 vorinitialisiert.

2.3.13 Feld-Variablen (dim)

```
DIM <var ><(><val|var|expr><)>
```

Mit Hilfe der Basic-Anweisungen *dim* ist es möglich einzelne Variablen als Felder zu deklarieren. Bsp.:

```
10 dim a(10)
20 a(1)=42
30 print a(1)
40 end
```

In Zeile 10 wird die Variable *a* als Feld mit 10 Elementen definiert. In den folgenden Zeilen wird auf das jeweils 2.Element des Feldes zugegriffen (Index 0...9). Die Elemente des Feldes sind immer vom Typ Integer.

Nach der Definition einer Variablen als Feld darf nur noch die Feld-Schreibweise der Variable, also z.B. *a(1)*, verwendet werden! Zugriffe auf Elemente außerhalb der mit *dim* definierten Größe werden mit einem Fehler quittiert. Ebenso wird bei der Definition eines Feldes verfahren, welches die Größe des verfügbaren Speichers übersteigen würde.

Anmerkung: Feldvariablen können nicht als Laufindex in einer for-next-Schleife verwendet werden.

2.3.14 Eingabe (input)

```
INPUT [str ;] <var > [[,] var]
```

Ermöglicht die interaktive Eingabe von Werten und deren Zuweisung zu Variablen. Bsp.:

```
10 input "a="; a
20 print a
30 input a, b
40 print a, b
50 end
```

Es können nur Zahlen eingeben werden (derzeit gibt es ja auch nur Integerwerte als Variablen). Bei anderen Eingaben wird der Wert 0 der entsprechenden Variablen zugewiesen.

Hinweis: Während des Inputbefehls befindet sich das BASIC-Programm in einer Eingabeschleife und steht an dieser Stelle. D.h. auch, dass parallel zum BASIC-Interpreter laufende Programmteile ebenfalls nicht abgearbeitet werden.

2.3.15 DATA/READ/RESTORE

Mit der DATA-Anweisung können konstante Werte im Quelltext hinterlegt werden, welche mit READ Variablen zugeweisen werden können. Mit RESTORE kann der Zeiger innerhalb der DATA-Liste auf die erste Stelle zurückgesetzt werden.

Ein Beispiel:

```
read a, b, c
print a, b, c
print "***"
restore
read a
print a
data 23, 24
data 0xff
end
```

2.3.16 EEPROM-Inhalt setzen (epeek)

epeek(<val|var|expr>)=<val|var|expr>

Der Wert in den Klammern gibt die Adresse im EEPROM an, deren Inhalt mit dem Wert rechts neben dem Istgleich gefüllt werden. Der Befehl ist AVR-spezifisch.

2.3.17 EEPROM-Inhalt lesen (epoke)

<var>=epoke(<val|var|expr>)

Kann auch selbst Element eines Ausdrucks sein. Der Wert in den Klammern gibt die Adresse im EEPROM an, deren Inhalt zurück gegeben werden soll. Der Befehl ist AVR-spezifisch.

2.3.18 Pause-Befehl (wait)

wait <val|var|expr>

Die Pausezeit ist in Millisekunden anzugeben. Intern wird _delay_ms() aus util/delay.h (mit allen Kon-

sequenzen) verwendet, also der Prozessor wartet wirklich und ist damit auch AVR-spezifisch.

2.3.19 I/O-Portrichtung (Ein- oder Ausgang) festlegen (dir)

```
dir(<port_str>, <pin>)=<val|var|expr>
```

port_str: AVR-Port „a“, „b“, „c“, „d“... (plattformspezifisch!)

pin: Pin-Nr. des Port (kann eine Zahl, eine Variable oder ein Ausdruck sein)

Es wird das angegebenen Pin des jeweiligen Ports als Aus- oder Eingang gesetzt. Dabei werden alle Werte >0 als Ausgang und Werte =0 wird als Eingang gewertet.

Der Befehl ist AVR-spezifisch. Welche Ports angesteuert werden können ist abhängig vom verwendeten Mikrocontroller und kann in ubasic_config.h eingestellt werden.

2.3.20 I/O-Port setzen (out)

```
out(<port_str>, <pin>)=<val|var|expr>
```

port_str: AVR-Port „a“, „b“, „c“, „d“... (plattformspezifisch!)

pin: Pin-Nr. des Port (kann eine Zahl, eine Variable oder ein Ausdruck sein)

Es wird das angegebenen Pin des jeweiligen Ports auf 0 oder 1 gesetzt. Dabei werden alle Werte größer 0 als 1 und Werte gleich 0 als 0 gewertet.

Der Befehl ist AVR-spezifisch. Welche Ports angesteuert werden können ist abhängig vom verwendeten Mikrocontroller und kann in ubasic_config.h eingestellt werden.

2.3.21 I/O-Port auslesen (in)

```
var=in(<port_str>, <pin>)
```

port_str: AVR-Port „a“, „b“, „c“, „d“... (plattformspezifisch!)

pin: Pin-Nr. des Port (kann eine Zahl, eine Variable oder ein Ausdruck sein)

Es wird das angegebene Pin des entsprechenden Ports ausgelesen und als Ergebnis zurückgegeben.

Der Befehl ist AVR-spezifisch. Welche Ports angesteuert werden können ist abhängig vom verwendeten Mikrocontroller und kann in ubasic_config.h eingestellt werden.

2.3.22 ADC-Auslesen (adc)

```
var=adc(<val|var|expr>)
```

Die Nummer in Klammern gibt den entsprechenden ADC an.

Der Befehl ist AVR-spezifisch. Die Konfiguration der möglichen ADC-Kanäle ist abhängig vom verwendeten Mikrocontroller und kann in ubasic_config.h eingestellt werden.

2.3.23 C-Routinen aufrufen (call)

```
[<var>=]call(str[, <val|var|expr>[, <val|var|expr>]])
```

str: Name der Funktion

Die folgenden Parameter sind die (konfigurierten) Übergabe der C-Funktion.

Aufruf von internen C-Routinen. Dazu muss in ubasic_call.* einige Voraussetzungen geschaffen werden, siehe auch entsprechendes Kapitel in diesem Dokument.

Funktioniert als Ausdruck, wie auch als Statement.

2.3.24 Zugriff auf interne C-Variablen (vpeek/vpoke)

Setzen einer internen C-Variable:

```
vpoke(<str>)=<val|var|expr>
```

Auslesen einer internen C-Variable:

```
<var>=vpeek(<str>)
```

str: Name der internen C-Variable

Bei *vpoke* wird die interne C-Variable auf den Wert nach dem Istgleich gesetzt. Der Rückgabewert von *vpeek* ist der Inhalt der internen C-Variable.

Siehe auch entsprechendes Kapitel in diesem Dokument.

2.3.25 Kommentar (rem)

```
rem [...]
```

Der nach *rem* folgende Text wird durch den Interpreter übersprungen und die Abarbeitung wird in der nächsten Basic-Programmzeile fortgesetzt.

2.4 Basic-Fehlerbehandlung

Soweit der Interpreter Fehler im Basic-Programm erkennt, werden sie in folgender Form auf der Standard-Ausgabe ausgegeben:

```
error <error-number> at sourceline: <linenum> (<basic-linenum>?) in program <name>
```

Dabei entspricht <linenum> der Zeile im BASIC-Quelltext (also nicht die BASIC-Zeilennumer) und <basic-linenum> der BASIC-Zeilennummer (soweit vorhanden). Wenn innerhalb der Konfiguration

der Support für externe Unterprogramme aktiviert ist, bezeichnet <name> die Programm-Datei, in der der Fehler aufgetreten ist.

<error-number> hat folgende Bedeutungen (siehe auch uBasic.h):

```
#define SYNTAX_ERROR                1
#define UNKNOWN_ADC_CHANNEL         2
#define UNKNOWN_IO_PORT             3
#define FOR_WITHOUT_TO              4
#define UNKNOWN_STATEMENT           5
#define UNKNOWN_CALL_FUNCT          6
#define UNKNOWN_CALL_FUNCT_TYP      7
#define UNKNOWN_CVAR_NAME           8
#define SHORT_IF_WITH_ELSE          9
#define GOSUB_STACK_DETH            10
#define FOR_STACK_DETH              11
#define GOSUB_STACK_INVALID         12
#define UNKNOWN_SUBPROC             13
#define GOSUB_NO_EXT_SUBPROC        14
#define ARRAY_OUT_OF_RANGE          15
#define OUT_OF_MEMORY               16
#define NOT_ENOUGH_DATA             17
#define UNKNOWN_LINENUMBER          18
#define INPUT_IS_NOT_NUMBER         19
#define UNKNOWN_VARIABLE            20
```

Für die richtige Differenzierung des Fehlers wird keine Garantie gegeben. Fakt ist aber, dass wenn ein Fehler auftritt, ist etwas am Basic-Programm faul (oder das Ende des dynamischen Speichers erreicht)!

Bei Auftreten eines Fehlers, wird das Basic-Programm abgebrochen.

3 Funktionsweise des Interpreters

Der uBasic-Interpreter unterteilt sich in zwei Komponenten:

- dem Tokenizer (tokenizer.*)
- dem eigentlichen uBasic (ubasic*.*)

Der Tokenizer analysiert den zu interpretierenden Code schrittweise nach ihm bekannten Syntaxelementen. Dazu sind in tokenizer.c diverse Schlüsselwörter (keywords[]) und Einzelzeichen (singlechar()) definiert, nach denen der Programmtext durchsucht wird. Weiterhin stellt tokenizer.c diverse Funktionen zur Verfügung, um das aktuelle Token abzufragen, die Tokenanalyse fortzuführen sowie bei einigen Token deren Wert zurückzugeben (String, Variable, Wert etc.).

In uBasic.c ist die vorgeschriebene Reihenfolge der einzelnen Tokenelemente, welche damit den eigentlichen Basic-Syntax ausmachen, und die daraus resultierenden Reaktionen, teilweise über mehrere Prozeduren verteilt, implementiert. Dabei wird hauptsächlich zwischen Statements (statement()) und Expression-Elementen (expr() -> term() -> factor()...) unterschieden.

Einige der Funktionen/Prozeduren werden rekursiv aufgerufen (vor allem expr() in ubasic.c). Dies bringt, gerade bei den beschränkten Ressourcen auf einem Mikrocontroller, die Gefahr mit sich, dass der Stack überlaufen könnte. Hautsächlich könnte dies bei sehr komplexen Expressions auftreten. D.h., also, dem Interpreter bei solchen Problemen nicht zu komplexe Basic-Ausdrücke mit vielen Klammern vorwerfen.

Durch diese sinnvolle und konsquente Aufteilung der Interpreter-Funktionen ist es relativ leicht möglich weiter Syntax-Elemente hinzuzufügen. Der "Durchlauf" durch die einzelnen Interpreter-Elemente ist leicht zu verstehen. Einfach mal den Quelltext lesen!

4 Einbinden in eigene Programme

4.1 Parametrisierung (ubasic_config.h)

Mit Hilfe diverser Defines in ubasic_config.h kann der Basic-Sprachumfang, der Speicherverbrauch sowie einige AVR-spezifische Dinge gesteuert werden. Folgende Defines sind definiert:

```
#define PRINTF(...)  usart_write(__VA_ARGS__)
```
Definiert die Standardausgabe für den Basic-Befehl *print*, im obigen Beispiel über die serielle Schnittstelle.

```
#define MAX_STRINGLEN 40
```
Maximale Länge eines Strings innerhalb des Basic-Befehls *print*. (Bsp.: maximal 40 Zeichen)

```
#define MAX_INPUT_LEN 11
```
Maximale Länge einer Eingabe bei BASIC-Befehl INPUT (Bsp.: maximal 11 Zeichen)

```
#define MAX_GOSUB_STACK_DEPTH 2
```
Maximale Schachtelungstiefe innerhalb von *gosub*-Anwesiungen. (Bsp.: maximale Schachtelungstiefe 2)

```
#define MAX_FOR_STACK_DEPTH 4
```
Maximale Schachtelungstiefe von *for-next*-Anweisungen. (Bsp.: maximale Schachtelungstiefe 4)

```
#define MAX_VARNUM 26
```
Maximale Anzahl von Basic-Variablen. Da Variablennamen nur aus einem Buchstaben bestehen können, sind maximal 26 Variablen möglich. (Bsp.: maximal 26 Variablen)

```
#define MAX_KEYWORD_LEN 8
```
Maximallänge der Schlüsselwörter in der Tabelle keywords[] (tokenizer.*). (Bsp.: maximal 8 Zeichen)

```
#define MAX_NAME_LEN 8
```
Maximale Länge von Funktions- und Variablennamen in *call*-, *vpeek*- und *vpoke*-Anweisungen über den die Variable bzw. Funktion via Basic angesprochen wird. (Bsp.: maximal 8 Zeichen)

```
#define UBASIC_ABS          1
#define UBASIC_NOT          1
#define UBASIC_XOR          1
#define UBASIC_SHL          1
#define UBASIC_SHR          1
```

```
#define UBASIC_CALL           1
#define UBASIC_CVARS          1
#define UBASIC_REM            1
#define UBASIC_PRINT          1
#define UBASIC_RND            1
#define UBASIC_INPUT          1
#define UBASIC_HEX_BIN        1
```
Über diese Defines kann der Sprachumfang des Basic-Interpreters beeinflusst werden. Ist der Wert mit 0 angegeben, steht das jeweilige Sprachelement nicht zur Verfügung. Die Define-Namen sind sprechend, mit CVARS sind die Basic-Befehle *vpeek* und *vpoke* für den Zugriff auf interne C-Variablen gemeint. Das Abwählen von nicht benötigten Basic-Elementen spart Speicherplatz.

```
#define UBASIC_EXT_PROC       1
```
Über dieses Define wird das Einbinden von externen Unterprogrammen mittels des *gosub*-Befehls gesteuert.

```
#define UBASIC_ARRAY    1
```
Variablen können als Felder definiert werden (Basic-Befehl *dim*).

```
#define UBASIC_DATA     1
```
Die BASIC-Befehle DATA/READ/RESTORE sind zulässig.

```
#define UBASIC_NO_LINENUM_ALLOWED   1
```
Es muss nicht vor jeder BASIC-Programmzeile eine Zeilennummer stehen.

```
#define USE_LINENUM_CACHE           1
#define MAX_LINENUM_CACHE_DEPTH     8
```
Es besteht die Möglichkeit häufig durch *goto* oder *gosub* angesprungene Basic-Zeilen intern zu puffern, um beim zweiten Ansprung dieser gepufferten Basic-Zeile selbige schneller zu finden und damit die Laufzeit zu verkürzen. Das obige erste Define schaltet diesen internen Puffer ein (Wert 1; aus mit Wert 0), mit dem zweiten Define legt man die maximale Anzahl der gepufferten Zeilen fest.

```
#define BREAK_NOT_EXIT  1
```
Mit diesem Define kann die Abbruchmethode bei Auftreten eines Fehlers im Basic-Programm (z.B. Syntax-Fehler) beeinflusst werden. Beim Wert 0 wird an den betreffenden Stellen im Code ein exit(1) ausgeführt und damit alles abgebrochen, also auch das Programm, in welches der Interpreter eingebettet ist. Beim Wert 1 wird nur die Abarbeitung des Basic-Programms abgebrochen, das Rahmenprogramm läuft weiter.

```
#define USE_AVR         1
```
Mit diesem Define können AVR-spezifische Basic-Anweisungen ein- (Wert 1) oder ausgeschaltet werden. Sämtliche folgenden Defines sind damit in Gänze deaktivierbar. Sind AVR-spezifische Dinge ausgeschaltet, sollte es möglich sein, den Basic-Interpreter auch auf anderen Plattformen einzusetzen.

```
#define USE_PROGMEM     1
```
Ist dieses Define mit dem Wert 1 belegt, werden einige statische Teile des Interpreters (z.B. die Tabelle

keywords[] in tokenizer.c) in den AVR-Programmspeicher verlagert und von dort auch gelesen. Damit wird der SRAM-Bereich des AVR entlastet. Zum Beispiel spart dieses gesetzte Define, bei allen eingeschalteten Sprachelementen, ca. 200 Byte des wertvollen SRAM-Bereiches, der sonst für die Tabelle keywords[] verwendet werden würde. Umgekehrt wächst allerdings der verbrauchte Platz im Programmspeicher, weil die PROGMEM-Leseroutinen etwas mehr Code generieren.

```
#define AVR_WAIT        1
#define AVR_EPEEK       1
#define AVR_EPOKE       1
#define AVR_DIR         1
#define AVR_IN          1
#define AVR_OUT         1
#define AVR_ADC         1
#define AVR_RND         1
```

Mit diesen Defines kann der Sprachumfang AVR-spezifischer Befehle gesteuert werden. Bei Wert 0 steht das entsprechende Sprachelement nicht zur Verfügung, was Speicherplatz im Programmspeicher und SRAM des AVR spart.

```
#define HAVE_PORTA      0
#define HAVE_PORTB      1
#define HAVE_PORTC      1
#define HAVE_PORTD      1
```

Mit diesen Defines wird festgelegt, welche I/O-Port mit den Basic-Befehlen *dir*, *in* und *out* gesteuert werden können. Grund für die diese Festlegung ist zum einen der verwendete Hardwarearchitektur des Mikrocontroller, nicht jeder Typ hat alle Ports. Oder zum anderen möchte man vielleicht bestimmte Ports von der Ansteuerung durch Basic-Befehle ausschließen.

```
#define ADC_COUNT_MAX   4
```

Gibt die Anzahl der, durch den Basic-Befehl adc abfragbaren ADC-Kanäle an.

4.2 Übersetzen, Testen

Prinzipiell ist es möglich, uBasic auf verschiedenen Plattformen einzusetzen. Im Quelltextpaket sind Testprogramme für AVR (main.c) und Linux (use-ubasic.c) enthalten. Entsprechende Makefiles sind ebenfalls zu finden (Makefile.avr, Makefile.linux).

Das AVR-Testprogramm nutzt die serielle Schnittstelle für die Ein- und Ausgaben. Weiterhin ist eine Reihe von Basic-Testprogrammen fest einkompiliert (siehe auch Kommentar in main.c).

Die Linux-Version erwartet das Basic-Programm als Datei, deren Name auf der Kommandozeile mitgegeben wird. Die Ausgaben erfolgen auf der Standardausgabe.

4.3 Einbinden, Start eines Basic-Programmes in eigene Applikationen

Das Einbinden des Interpreters in eigene Programme geht relativ einfach. Prinzipiell ist nur uBasic.h entsprechend zu includieren. Ggf. sind die Defines PRINTF (in ubasic_config.h für Basic-Befehl *print*) und DEBUF_PRINTF (für evt. Debug-Ausgaben) in tokenizer.c, ubasic.c und uBasic_call.c an die eigene Ausgaberoutinen anzupassen. Des weiteren kann man in ubasic_config.h teilweise den Basic-Sprachumfang steuern (siehe weiter unten).

Das abzuarbeitende Basic-Programm muss in dem Char-Array stehen, welches der Prozedur uBasic_init() übergeben wird. Wie das Programm in dieses Char-Array gelangt, ist jedem selbst überlassen. Die einzelnen Basic-Zeilen in dem Array müssen mit '\n' (0x0A) abgeschlossen sein.

Zur Abarbeitung des Basic-Programmes ist ungefähr folgender Konstrukt im eigenen Programm einzubauen:

```
uBasic_init(program);
do {
    uBasic_run();
   } while(!uBasic_finished());
```

uBasic_init() setzt einen internen Pointer auf den Anfang des Programmtextes und initialisiert ein paar weitere interne Variablen.

Die folgende do-while-Schleife wird solange abgearbeitet, bis das Basic-Programm endet, wobei uBasic_run() jeweils immer genau eine Basic-Zeile abarbeitet. Es ist also z.B. denkbar, in dieser Schleife auch noch das zu tun/aufzurufen, was wärend des Basic-Programmlaufes "parallel" im Mikrocontroller abgearbeitet werden soll. Anmerkung: es kann aber kein zweites Basic-Programm parallel laufen!

Eine weitere Variante, um die Abarbeitung des Basic-Programmes in einer bestehenden Hauptschleife einzubauen, könnte ungefähr so aussehen:

```
while (1) {
   ...
   // irgendwo Programm laden und uBasic_init(program) aufrufen
   ...
   if (!uBasic_finished) uBasic_run;
   ...
}
```

Als Referenz für das Einbinden des Basic-Interpreters in eigene Programme, wird das Studium von main.c, welche im Quellcode-Archiv enthalten ist, angeraten.

4.4 Schnittstelle zu anderen internen C-Funktionen und -Variablen

4.4.1 Basic-Befehl CALL

Der *call*-Befehl ermöglicht eine drastische Erweiterung des Sprachumfangs von uBasic. Mit Hilfe die-

ses Mechanismus können (fast) beliebige C-Funktionen des Programms angesprochen werden, in welches der Interpreter eingebunden ist. Dabei ist die Anzahl Übergabeparameter, deren Typ sowie der Rückgabewert der Funktion für uBasic frei einstellbar.

Dazu sind allerdings einige Dinge in den Quelltext-Dateien ubasic_call.h und ubasic_call.h vorzubereiten.

Das folgende Erläuerung bezieht sich auf die Einbindung folgender C-Funktionen im Basic-Interpreter und deren Aufrufmöglichkeit mit dem Basic-Befehl *call*:

```
void a(void) {
      DDRB |= (1 << PB1);
      PORTB |= (1 << PB1);
}

void b(int p1) {
      DDRB |= (1 << PB1);
      if (p1) PORTB |= (1 << PB1); else PORTB &= ~(1 << PB1);
}

int c(int p1) {
      int r=0;
      ADMUX =  p1;
      ADMUX |= (1<<REFS0);
      ADCSRA = (1<<ADEN) | (1<<ADPS1) | (1<<ADPS0);
      ADCSRA |= (1<<ADSC);
      while (ADCSRA & (1<<ADSC));
      r = ADCW;
      ADCSRA |= (1<<ADSC);
      while (ADCSRA & (1<<ADSC));
      r = ADCW;
      ADCSRA=0;
      return r;
}
```

Dabei ist a() eine Prozedure ohne Übergabeparameter und Rückwert, b() eine Prozedure mit einem Übergabeparameter aber ohne Rückgabewert und c() mit einem Übergabeparameter und einem Rückgabewert.

ubasic_call.h:

```
(1)
// Funktionspointertypen
#define VOID_FUNC_VOID        0
#define VOID_FUNC_INT         1
#define INT_FUNC_INT          2

(2)
// Strukturdefinition fuer Funktionspointertabelle
typedef struct {
      char* funct_name;
      union ftp {
        void (*VoidFuncVoid)(void);
        void (*VoidFuncInt)   (int);
        int  (*IntFuncInt)    (int);
      } funct_ptr;
      unsigned char typ;
} callfunct_t;
```

(1) Es empfiehlt sich für jeden „Funktionstyp“ ein eigenes Define anzulegen. Mit „Funktionstyp“ ist die jeweilige Anzahl/Typ der Übergabeparameter und ob es einen Rückgabewert gibt, gemeint. Es reicht ein Define für unterschiedliche C-Funktionen gleichen Typs.

(2) Für jeden neuen „Funktionstyp“ ist in der Strukturdefinition callfunct_t ein entsprechender Eintrag in der union-Definition ftp aufzunehmen. Einige Beispiele sind im obigen Quellcodefragment aufgeführt.

ubasic_call.c:

```
(3)
// Funktionspointertabelle
callfunct_t callfunct[] = {
    {"a",   .funct_ptr.VoidFuncVoid=a,    VOID_FUNC_VOID},
    {"b",   .funct_ptr.VoidFuncInt=b,     VOID_FUNC_INT},
    {"c",   .funct_ptr.IntFuncInt=c,            INT_FUNC_INT},
    {NULL, {NULL},      255}
};

...

(4)
// je nach Funktionstyp (3.Spalte in Funktionspointertabelle)
// Parameterliste aufbauen und Funktion aufrufen
switch (callfunct[idx].typ){
      case VOID_FUNC_VOID:
      callfunct[idx].funct_ptr.VoidFuncVoid();
            break;
      case VOID_FUNC_INT:
            accept(TOKENIZER_COMMA);
            p1=expr();
            callfunct[idx].funct_ptr.VoidFuncInt(p1);
            break;
      case INT_FUNC_INT:
            accept(TOKENIZER_COMMA);
            p1=expr();
            r=callfunct[idx].funct_ptr.IntFuncInt(p1);
            break;

...

(5)
// bei Funktionspointertypen ohne Rueckgabewert ein Token
// weitergelesen...
if  ((callfunct[idx].typ == VOID_FUNC_VOID) ||
      (callfunct[idx].typ == VOID_FUNC_INT)
      ) tokenizer_next();
```

(3) In der Tabelle callfunct[] sind die einzelnen, durch uBasic aufrufbaren C-Funktionen einzufügen. Dabei entspricht die 1.Spalte dem Namen der Funktion, mit dem die Funktion mittels des uBasic-Befehl *call* aufgerufen werden soll. Die 2.Spalte nimmt den Zeiger auf die C-Funktion auf, wobei aber unbedingt darauf zu achten ist, dass die entsprechend korrekte Typdefinition aus der union ftp (uBasic_call.h) verwendet wird. Das gleiche gilt für die 3.Spalte, die das entsprechende Define für den Funktionstyp aufnimmt.

(4) In dem switch-Block in der Prozedure call-statement() muss für jeden Funktionstyp ein entsprechender case-Zweig vorhanden sein, in dem die jeweiligen Parameter des *call*-Befehles (uBasic) via Tokenizer-Aufrufe ausgelesen werden, der entsprechende Funktionsaufruf über den jeweiligen Funktionspointer erfolgt sowie der u.U. vorhandene Rückgabewert verarbeitet wird.

(5) Bei C-Funktionen ohne Rückgabewert, der auch an den call-Befehl (uBasic) zurück gereicht werden soll, ist das entsprechende Funktionstyp-Define in der if-Anweisung (call_statement()), mit ODER verknüpft, aufzunehmen.

Selbstverständlich sind die Header-Dateien, in denen die entsprechenden C-Variablen definiert sind, zu includieren.

4.4.2 Basic-Befehle VPEEK, VPOKE

Mit den Basic-Anweisungen *vpeek* und *vpoke* ist es möglich auf interne C-Variablen der Rahmen-Programms zuzugreifen, in das der Interpreter eingebettet ist. Dazu sind in ubasic_cvars.c diese Variablen dem Interpreter über die dort definierte Tabelle cvars[] bekannt zumachen.

Beispiel (ubasic_cvars.c):

```
// Variablenpointertabelle
cvars_t cvars[] = {
    {"a", &va},
    {NULL, NULL}
};
```

In dem Beispiel wird die interne C-Variable va vom Typ Integer wird mit dem Namen „a“ verknüpft. Über diesen Namen kann mit den Basic-Befehlen *vpeek* und *vpoke* auf den Inhalt der Variablen zugegriffen werden. Es sind nur Integervariablen einbindbar, da der Basic-Interpreter nur mit Integer arbeiten kann.

Selbstverständlich sind die Header-Dateien, in denen die entsprechenden C-Variablen definiert sind, zu includieren.

5 Variabler Zugriff auf Basic-Quelltext

5.1 Ziel

Ziel der in der Folge beschrieben Systematik ist es, das Einlesen des Basic-Quelltextes von einem beliebigen Medium zu ermöglichen. Medien könnten z.B. sein:

- ein Array innerhalb des (Haupt-)Programms (so wie in der Ursprungsversion mit dem Nachteil, dass die Größe des Arrays konstant ist und von Anfang an den frei verfügbaren dynamischen Speicher entsprechend reduziert; die max. Größe eines Basic-Programmes wird durch die Dimension dieses Arrays begrenzt)
- Flash-RAM, EEPROM eines Mikrocontrollers, soweit vorhanden (z.B. AVR-Mikrocontroller)
- angeschlossene externe Speicherbausteine
- ein eventuell vorhandenes Filesystem (SD-Karte, Festplatte usw.)

- usw.

Voraussetzung ist, dass innerhalb des Quelltext-Mediums, mittels geeigneter Funktionen und berechenbarer Zeiger, frei und in jede Richtung positioniert werden kann, da ja der Quelltext-Parser (tokenizer.c) nicht nur sequenziell arbeitet, sondern, je nach Kontext, auch Programmtext überspringen oder im selbigen zurückspringen (goto, gosub, for-next usw.) sowie neu analysieren muss. Bei uBasic handelt es sich halt um einen Interpreter, der den Quelltext zur Laufzeit analysiert und unmittelbar ausführt.

5.2 Umsetzung

Der eigentliche Zugriff auf den Basic-Quelltext erfolgt innerhalb des Parsers (tokenizer.c) mittels eines Zeigers, der immer auf die aktuelle Position innerhalb des Basic-Programmes zeigt. Folgende Funktionen müssen mit diesem Zeiger realisierbar sein:

- Abfrage der Zeigersposition
- Abfrage des Wertes der Speicherzelle, auf die der Zeiger aktuell zeigt
- Inkrementieren des Zeigers um eine Stelle weiter
- Setzen einer beliebigen neuen Zeigerposition
- Basic-Programmende-Erkennung

Sämtliche relevanten Stellen im uBasic-Quelltext sind mit Defines realisiert, deren Umsetzung je nach Medium-Zugriffsart neu definiert werden können. Die entsprechenden Definition erfolgen in der Header-Datei tokenizer_access.h und haben folgende Bedeutungen (mit jeweils dem Beispiel für den Zugriff auf einen definierten Bereich im Speicher):

```
#define PTR_TYPE char const *
```
Definition des Types der Variable, die den Zeigerwert beinhaltet. Könnte theoretisch auch vom Typ Integer o.ä. sein, wenn die jeweilige Zugriffsart dies erfordert.

```
static PTR_TYPE ptr;
```
Definition der Zeigervariable selbst.

```
#define PROG_PTR ptr
```
Bezeichnet die Zeigervariable im Quelltext.

```
#define SET_PROG_PTR_ABSOLUT(param) (PROG_PTR = (param))
```
Setzt den Zeiger auf die Position param (absolut zum Anfang). Wichtig ist, dass auch der physische Zeiger des entsprechenden Mediums danach auf die korrekte Position zeigt bzw. der Inhalt der referenzierten Speicherzelle mittels GET_CONTENT_PROG_PTR ermittelbar ist.

```
#define GET_CONTENT_PROG_PTR *ptr
```
Gibt den Inhalt der Speicherzelle zurück, auf die der Zeiger zeigt. Hinweis: Routinen wie z.B. fgetc() für einen Dateizugriff aus stdio.h inkrementieren automatisch ihren internen Zeiger auf die nächste

Stelle in der Datei!

```
#define INCR_PROG_PTR ++ptr
```
Erhöht den Wert des Zeiger um eine Stelle. Wichtig ist, dass auch der physischen Zeiger des entsprechenden Mediums danach auf die korrekte Position zeigt und der Inhalt der referenzierten Speicherzelle mittels GET_CONTENT_PROG_PTR abfragbar ist.

Theoretisch könnte auch SET_PROG_PTR_ABSOLUT verwendet werden:

```
#define INCR_PROG_PTR (SET_PROG_PTR_ABSOLUT(PROG_PTR+1))
```
Aber vielleicht gibt es, je nach Medium, optimalere Operationen (wie z.B. bei Zugriffen via fseek()/fgetc() innerhalb eines Dateisystems). Aus diesen Grund ist dieses Define extra herausgeführt.

```
#define END_OF_PROG_TEXT GET_CONTENT_PROG_PTR == 0
```
Bedingung mit der das physische Ende des Basic-Programmtextes erkannt werden kann. Das Ergebnis muss wahr (größer 0) sein, wenn das Programmende erreicht ist.

Im Quelltextarchiv sind Beispiele für unterschiedliche Zugriffsmethoden auf den Basic-Quelltext zu finden, um die Funktionsweise verdeutlichen:

- Programm im Flash-Speicher eines AVR-Mikrocontrollers
- Programm auf einer SD-Karte an einem AVR-Mikrocontroller
- über direkte Dateisystemzugriffe unter Linux

6 Aufruf von externen Unterprogrammen

Innerhalb des *gosub*-Befehles ist es möglich, statt einer Programmzeile innerhalb des aktuellen Programms, auch ein weiteres externes Programm als String anzugeben. Beispiel:

```
10 GOSUB "UP1"
```
Dieses Programm (hier UP1) wird im Fall des Aufrufs geöffnet, von der ersten Zeile abgearbeitet und bei Auftreten eines *return* wieder verlassen, um die Kontrolle dem aufrufenden Programm zurückzugeben. Sämtliche Variableninhalte usw. bleiben über die Programmgrenzen hinaus erhalten, wirken also wie „globale Variablen“. Das gleiche gilt für sämtliche Stack- und Cache-Bereiche (for-next, gosub, Zeilennummern-Cache) des Hauptprogrammes.

Die Umschaltung in den jeweiligen Programmkontextes erfolgt mit der Prozedur *switch_proc()* in *ubasic_ext_proc.c*. Diese Prozedur muss für das jeweilige Speichermedium angepasst werden. Prinzipiell sind jeweils folgende Schritte zu implementieren:

- Schließen des derzeitig geöffneten Programms;
- Öffnen des Programms dessen Name als Parameter übergeben wurde
- Setzen der Zeiger *PROG_PTR* und *programm_ptr* auf den Anfang des neuen Programms;
- Aufruf *tokenizer_init()*
- Setzen der Variable *current_proc* auf aktuellenProgrammname
- Fehlerbehandlung nicht vergessen, z.B. Programm nicht vorhanden

Ob der Aufruf von externen Unterprogrammen möglich ist, wird über das Define

```
#define UBASIC_EXT_PROC       1
```

gesteuert.

Das Define

```
#define MAX_PROG_NAME_LEN     8
```

gibt die maximale Länge des Programmnamen an.

Beispiele für verschiedene Speichermedien sind im Quelltextarchiv enthalten.

7 Formale Syntax-Beschreibung (Erweiterte Backus-Naur-Form)

```
Line           = [number | number ":"] [statement] "\\n";
statement      = PRINT printlist |
                 [LET] var "=" expression |
                 GOTO expression |
                 GOSUB (expression | characterstr) |
                 RETURN |
                 IF expression relop expression (THEN statement [ELSE statement]|[THEN] statement) |
                 FOR var "=" expression ( TO | DOWNTO ) expression [STEP expression]|
                 NEXT var |
                 REM { any_character } |
                 DIM var "(" expression ")" |
                 INPUT [characterstr ";"] var {["," ] var} |
                 DATA number {["," ] number} |
                 READ var {["," ] var} |
                 RESTORE |
                 SRAND |
                 WAIT expression |
                 OUT "(" characterstr "," expression ")" "=" expression |
                 EPOKE "(" expression ")" "=" expression |
                 VPOKE "(" characterstr ")" "=" expression |
                 DIR "(" characterstr "," expression ")" "=" expression |
                 CALL "(" characterstr "," exprlist ")";
printlist      = [printitem | printitem separator printlist];
printitem      = (expression | characterstr);
varlist        = var | var "," varlist;
exprlist       = expression | expression "," exprlist;
expression     = ["-"|"+"] (term | "(" term ")");
term           = factor | factor operator term;
factor         = var | ["+" | "-"] number | expression | function;
function       = RAND "(" expression ")" |
                 ABS "(" expression ")" |
                 NOT "(" expression ")" |
                 EPEEK "(" expression ")" |
                 ADC "(" expression ")" |
                 IN "(" characterstr "," expression ")" |
                 CALL "(" characterstr "," exprlist ")" |
                 VPEEK "(" characterstr ")"  ;
number         = decnumber | hexnumber | dualnumber;
decnumber      = decdigit {decdigit};
hexnumber      = "0" ("x"|"X") hexdigit {hexdigit};
dualnumber     = "0" ("b"|"b") dualdigit {dualdigit};
separator      = "," | ";";
var            = ("A" | "..." | "Z" | "a" | "..." | "z") ["(" expression ")"];
decdigit       = "0" | "..." | "9";
hexdigit       = "0" | "..." | "9" | "A" | "..." | "F" | "a" | "..." | "f";
dualdigit      = "0" | "1";
relop          = "<" | ">" | "=" | "<=" | ">=" | "<>" | "<>";
operator       = "+" | "-" | "*" | "/" | "%" | "mod" | "|" | "or" | "&" |
                 "and" | "xor" | "shl" | "shr";
characterstr   = "\""  { any_character } "\"";
```

Mit diversen Tools können aus obiger EBNF-Beschreibung die bekannten Syntax-Diagramme generiert werden. Ein Online-Tool ist z.B. unter http://www-cgi.uni-regensburg.de/~brf09510/syntax.html zu finden.

8 Bekannte Fehler

- Derzeit sind keine Fehler bekannt.

Sollten weitere Fehler gefunden werden, können sie gern über die unten angegebene E-Mailadresse gemeldet werden.

9 Kontakt

Uwe Berger; bergeruw@gmx.net

Webseiten zum Projekt: http://www.mikrocontroller.net/articles/AVR_BASIC