Dokumentation uBasic (auf einem AVR)

Uwe Berger (bergeruw@gmx.net)

Stand: 10.06.2010

Inhalt

1Motivation....3
2uBasic-Eigenschaften....3
3Sprachumfang....3
3.1Generell....3
3.2Einschränkungen....4
3.3uBasic-Syntax....4
3.3.1Schleife von/bis (for/to/next)....4
3.3.2Bedingte Anweisung (if/then/else)....5
3.3.3Sprung (goto)....5
3.3.4Unterprogramm aufrufen (gosub)....5
3.3.5Programm-Ende (end)....5
3.3.6Zufallswert (srand/rand)....5
3.3.7absoluter Betrag (abs)....6
3.3.8Complement (not)....6
3.3.9Ausgabe (print)....6
3.3.10Variable setzen (let)....6
3.3.11EEPROM-Inhalt setzen (epeek)....6
3.3.12EEPROM-Inhalt lesen (epoke)....7
3.3.13Pause-Befehl (wait)....7
3.3.14I/O-Portrichtung (Ein- oder Ausgang) festlegen (dir)....7
3.3.15I/O-Port setzen (out)....7
3.3.16I/O-Port auslesen (in)....8
3.3.17ADC-Auslesen (adc)....8
3.3.18C-Routinen aufrufen (call)....8
3.3.19Zugriff auf interne C-Variablen (vpeek/vpoke)....8
3.3.20Kommentar (rem)....9
3.4Basic-Fehlerbehandlung....9
3.5Basic-Beispielprogramme....9
3.5.1for-next, gosub-return, print....9
3.5.2abs(), not()....10

3.5.3Zufallswert....10
3.5.4Kommentar....10
3.5.5Call-Befehl....10
3.5.6VPOKE, VPEEK....11
3.5.7EEPROM setzen/lesen....11
3.5.8Endlosschleife mit wait-Befehl....11
3.5.9I/O-Port auslesen....11
3.5.10ADC auslesen....11
3.5.11Blinklichter....11
Variante 1....11
Variante 2....12
Variante 3....12
4Funktionsweise des Interpreters....12
5Einbinden in eigene AVR-Programme....13
5.1Parametrisierung....13
5.2Einbinden, Start eines Basic-Programmes in eigene Applikationen....13
5.3Schnittstelle zu anderen internen C-Funktionen und -Variablen....14
5.3.1Basic-Befehl CALL....14
5.3.2Basic-Befehle VPEEK, VPOKE....16
6Kontakt....16

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 Kompiler 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 designden 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 zumindestens 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 AVR-uBasic für AVRs.

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

2 uBasic-Eigenschaften

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 einbindbar
- kleiner übersetzter Code, geringer Speicherverbrauch
- Sprachumfang und Speicherverbrauch parametrisierbar

3 Sprachumfang

3.1 Generell

Basic-Zeilen fangen immer 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!

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.

3.2 Einschränkungen

Folgende Einschränkungen gelten für Basic-Programme

- 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)
- Schachtelungstiefe for-Befehl: ist begrenzt und via define einstellbar (uBasic_config.h)
- Schachtelungstiefe gosub-Befehl: ist begrenzt und via define einstellbar (uBasic_config.h)
- zulässige Vergleichsoperatoren: <, >, =
- Länge des Basic-Quelltext: ist begrenzt durch vorhandene Speichergrößen
- es gibt keine Arrays

3.3 uBasic-Syntax

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

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

...

next <var>

Schachtelungen sind möglich.

3.3.2 Bedingte Anweisung (if/then/else)

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

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.

3.3.3 Sprung (goto)

goto <val>

val gibt eine gültige Zeilennummer an, zu der im Kontext gesprungen werden soll.

3.3.4 Unterprogramm aufrufen (gosub)

gosub <val>

...

return

val gibt eine gültige Zeilennummer an, an der das Unterprogramm beginnt. Ende-Anweisung im Unterprogramm ist *return*, es wird zur Zeile nach dem entsprechenden *gosub*-Befehl zurückgesprungen und die Abarbeitung des Programms fortgesetzt.

Schachtelungen sind möglich.

3.3.5 Programm-Ende (end)

end

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

3.3.6 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.

3.3.7 absoluter Betrag (abs)

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

Kann auch selbst Element eines Ausdrucks sein.

3.3.8 Complement (not)

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

Kann auch selbst Element eines Ausdrucks sein.

3.3.9 Ausgabe (print)

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

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

3.3.10 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.

3.3.11 EEPROM-Inhalt setzen (epeek)

epoke(<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.

3.3.12 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.

3.3.13 Pause-Befehl (wait)

wait <val|var|expr>

Die Pausezeit ist in Millisekunden anzugeben. Intern wird _delay_ms() aus util/delay.h verwendet, also der Prozessor wartet wirklich und ist damit auch AVR-spezifisch.

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

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

port: AVR-Port a,b,c,d... (plattformspezifisch!)

pin: Pin-Nr. des Port

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.

3.3.15 I/O-Port setzen (out)

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

port: AVR-Port a,b,c,d... (plattformspezifisch!)

pin: Pin-Nr. des Port

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.

3.3.16 I/O-Port auslesen (in)

var=in(<port>, <pin>)

port: AVR-Port a,b,c,d... (plattformspezifisch!)

pin: Pin-Nr. des Port

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.

3.3.17 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.

3.3.18 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.

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

Setzen einer internen C-Variable:

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

Auslesen einer internen C-Variable:

<var>=epeek(<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.

3.3.20 Kommentar (rem)

rem [...]

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

3.4 Basic-Fehlerbehandlung

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

error <error-number> at line: '<line-number>'

Dabei ist <line-number> die Basic-Zeile, in der der Fehler aufgetreten ist.

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

- 1 → allgemeiner Syntax-Fehler
- 2 → AVR-Erweiterung; unbekannter ADC-Kanal; uBasic_config.h
- 3 → AVR-Erweiterung; unbekannter I/O-Port; uBasic_config.h
- 4 → FOR-Schleife ohne TO oder DOWNTO
- 5 → unbekannter Befehl (Statement)
- 6 → CALL-Befehl: unbekannter Funktionsname
- 7 → intern; CALL-Befehl: unbekannter Funktionspointertyp
- 8 → VPEEK/VPOKE-Befehl: unbekannter Variablenname

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.

3.5 Basic-Beispielprogramme

3.5.1 for-next, gosub-return, print

```
10  gosub 100
15  a=30
20  for i = 1 to a step 10
```

```
30  print "i=", i
40  next i
50  end
100 print "subroutine"
110 return
```

3.5.2 abs(), not()

```
10 print abs(-8)
20 print not(7)
30 end
```

3.5.3 Zufallswert

```
10  srand
20  for v = 1 to 2000
30  y=rand(9)
40  if y=0 then a=a+1
50  if y=1 then b=b+1
60  if y=2 then c=c+1
70  if y=3 then d=d+1
80  if y=4 then e=e+1
90  if y=5 then f=f+1
100 if y=6 then g=g+1
110 if y=7 then h=h+1
120 if y=8 then i=i+1
130 if y=9 then j=j+1
140 next v
150 print a,b,c,d,e,f,g,h,i,j
160 end
```

3.5.4 Kommentar

```
10 rem Das ist ein Kommentar
20 print "Hallo..."
30 end
```

3.5.5 Call-Befehl

```
10 call("a")
20 print call("c",0)
30 wait 500
40 call("b", 0)
50 wait 500
60 goto 10
70 end
```

3.5.6 VPOKE, VPEEK

```
10 vpoke("a")=123
20 a=vpeek("a")
30 end
```

3.5.7 EEPROM setzen/lesen

```
10 epoke(2)=55
20 epoke(3)=11
30 a=epeek(2)
40 print a
50 print epeek(3)+3
60 end
```

3.5.8 Endlosschleife mit wait-Befehl

```
10 a=1
20 print a
30 wait 1000
40 a=a+1
50 goto 20
60 end
```

3.5.9 I/O-Port auslesen

```
10 print in(b,0)
20 end
```

3.5.10 ADC auslesen

```
10 print adc(0)
20 end
```

3.5.11 Blinklichter

Variante 1

```
10 dir(b,1)=1
20 out(b,1)=1
30 wait 1000
40 out(b,1)=0
50 wait 1000
60 goto 20
70 end
```

Variante 2

```
10 dir(b,1)=1
20 a=0
30 out(b,1)=a
40 if a=1 then a=0 else a=1
50 wait 1000
60 goto 30
70 end
```

Variante 3

```
10 dir(b,1)=1
20 a=0
30 a=a+1
40 if (a%2)=1 then out(b,1)=0 else out(b,1)=1
50 wait 1000
60 goto 30
70 end
```

4 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!

5 Einbinden in eigene AVR-Programme

5.1 Parametrisierung

Mit Hilfe diverser Defines in ubasic_config.h kann der Basci-Sprachumfang, der Speicherverbrauch sowie einige AVR-spezifische Dinge gesteuert werden. Die einzelnen Defines in dieser Header-Datei sind dort entsprechend kommentiert.

5.2 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, ist das Studium von main.c, welche im Quellcode-Archiv enthalten ist, angeraten.

5.3 Schnittstelle zu anderen internen C-Funktionen und -Variablen

5.3.1 Basic-Befehl CALL

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

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 mit 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.

5.3.2 Basic-Befehle VPEEK, VPOKE

Mit den Basic-Anweisungen *vpeek* und *vpoke* ist es möglich auf interne C-Variablen der Programmes 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}
};
```

Die interne C-Variable va wird mit dem Namen „a“ verknüpft. Über diesen Namen kann mit den Basic-Befehlen *vpeek* und *vpoke* auf den Inhalt der Variablen zugegriffen werden.

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

6 Kontakt

Uwe Berger; bergeruw@gmx.net

Webseite zum Projekt:
http://www.bralug.de/wiki/Basic-Interpreter_f%C3%BCr_AVR_%28uBasic-avr%29