uBasic Port für STM32F4 von Uwe Becker

Das original „uBasic“ von Adam Dunkels findet sich unter :
„http://dunkels.com/adam/ubasic/“

Die „Minimal“-Version vom uBasic-Port für den STM32F4 beinhaltet folgendes :

Basic-Befehle :

'REM'	Kommentarzeile
'LET'	Variablenzuweisung
'PRINT'	Ausgabe (per UART)
'IF/THEN/ELSE'	Bedingte Anweisung
'FOR/TO/STEP/NEXT'	Schleife
'GOTO'	Sprung
'GOSUB/RETURN'	Sprung zu Unterprogramm und Rücksprung
'END'	Ende vom Basic-Programm
'RIGHT$'	Liefert die ersten 'n' Zeichen von einem String
'ABS'	Liefert den Absolutwert einer Zahl

Operatoren :

'+'	Addition
'-'	Subtraktion
'*'	Multiplikation
'/'	Division
'%'	Modulo
'&'	Bitweises UND
'|'	Bitweises ODER
'^'	Bitweises XOR

Vergleichsoperatoren :

'='	Gleichheit
'<'	Kleiner
'>'	Größer

Variabeln :

'a' bis 'z'	für Integer-Zahlen
'a$' bis 'z$'	für Strings

Zahlenschreibweise :

'123'	positive Zahl
'-123'	negative Zahl
'0xA3F'	positive Zahl in Hex (A3Fh = 2623d)

Klammersetzung :

a=((5+3)*2)-2	a=14

Benutzung der Files im eigenen Projekt :

Files :

Im eigenen Projekt müssen die 4 Files
„ubasic.c“ + „ubasic.h“
„tokenizer.c“ + „tokenizer.h“

hinzugefügt werden.

Basic-RAM :

es muss ein Globales RAM-Array eingerichtet werden in dem später
das Basic-Programm liegen wird.

z.B. „char basic_ram[1024]; // 1k für Basic-Programm“

Basic-Program :

das Basic-Programm muss in das RAM-Array kopiert werden.
Die Quelle vom Basic-Programm (Flash/SD-Karte/UART usw)
ist von der eigenen Implementation abhängig.

Init vom uBasic :

zur Initialisierung muss die Funktion „ubasic_init“ aufgerufen werden
mit einem Pointer auf das Basic-RAM

z.B. „ubasic_init(basic_ram,0);“

Start vom Basic-Program :

nach der Initialisierung muss in einer Schleife die Funktion „ubasic_run“
solange aufgerufen werden, bis das Ende vom Basic-Programm erreicht ist :

z.B.

```
do {
  ubasic_run();
} while(!ubasic_finished());
```

Minimal Basic-Programm :

```
10 REM Testprogram
20 FOR n = 1 TO 10
30 PRINT n
40 NEXT n
50 END
```

Hinweise zu meiner Version von uBasic

ich habe das uBasic von Adam Dunkels erweitert und für mich angepasst
(ein Filevergleich mit dem Original zeigt die Unterschiede)

Neben der Beseitigung von ein paar Bugs habe ich noch folgendes hinzugefügt :

1. Der Print-Befehl wird auf eine UART umgeleitet (kann im H-File deaktiviert werden)
2. String Verarbeitung über String Variabeln 'a$' bis 'z$'
3. Zahlenformat HEX : '0x0' bis '0xFFFF'
4. Negative Zahlen möglich
5. Basic-Befehl 'REM' zum einfügen von Kommentarzeilen
6. Pre-Parser vom Basic-Programm der die Geschwindigkeit bei der späteren abarbeitung deutlich erhöht.
   > alle Basic-Befehle werden in Bytecodes gewandelt
   (beim Original mussten alle Zeichen vom Basic-Befehl geprüft werden)
   > alle Sprungziele von GOTO/GOSUB/FOR werden in einem Array gesammelt und die direkte RAM-Adresse wird ermittelt und gespeichert
   (beim Original wird jedesmal das ganze Programm nach der Zeilen-Nr durchsucht)
7. Basic-Befehl 'FOR' kann mit dem Parameter 'STEP' erweitert werden
8. Auch rückwärts laufende Schleifen möglich
9. Basic-Befehle können verschachtelt werden
10. Der Basic-Befehl "ABS" dient als Vorlage für Befehle mit Rückgabewert=<int>
11. Der Basic-Befehl "RIGHT$" dient als Vorlage für Befehle mit Rückgabewert=<string>
12. Der Basic-Befehl "REM" dient als Vorlage für Befehle mit Rückgabewert=<void>

Einschränkungen :

Einige Einschränkungen vom Basic bleiben allerdings bestehen :

1. Jede Zeile muss mit einer eindeutigen Zeilen-Nummer beginnen
2. Sprungziele bei GOTO/GOSUB dürfen keine Variablen sein
3. Das Basic-Program muss in einem RAM-Bereich liegen
4. Keine Float Unterstützung
5. Nur 26 Integer- und 26 String-Variablen nutzbar
6. Systembedingt keine sehr hohe Abarbeitungsgeschwindigkeit

Neuer Basic-Befehl hinzufügen : (return_value = <int>)

Hier eine Schritt für Schritt Anleitung um einen neuen Basic-Befehl hinzuzufügen :

Als Beispiel soll der Befehl „ASC“ hinzugefügt werden
Der Befehl liefert den Ascii-Wert eines Zeichens
Syntax : ASC(<str_expr>) Return_Wert : <int>

1. File : „tokenizer.h“ ein neues Token am Ende der enum Liste hinzufügen
   Code : „TOKENIZER_ASC“
2. File : „tokenizer.c“ das neue Token in der Keywordliste hinzufügen
   Code : {"ASC", TOKENIZER_ASC},
3. File : „tokenizer.c“ das neue Token in der Funktion „tokenizer_set_expression“ hinzufügen und zwar bei denen mit dem Rückgabewert <int>,
   Code : case TOKENIZER_ASC :
4. File : „ubasic.c“ einen neuen Funktionsprototyp hinzufügen
   Code : static int cmd_asc(void);
5. File : „ubasic.c“ die neue Funktion in der Funktion „factor(void)“ als neuen 'Case' hinzufügen
   Code :
```
case TOKENIZER_ASC : // 'ASC'
        r=cmd_asc();
break;
```
6. File : ubasic.c“ unter dem Abschnitt „NEW <INTEGER> STATEMENTS BY UB“ die neue Funktion ausprogrammieren
   Code :
```
static int cmd_asc(void)
{
 char *s;
 int ret_value=0;

 accept(TOKENIZER_ASC);
 accept(TOKENIZER_LEFTPAREN);
 s=strexpr();
 accept(TOKENIZER_RIGHTPAREN);
 ret_value=s[0]; // ascii wert vom ersten Zeichen ermitteln

 return ret_value;
}
```

FERTIG

Neuer Basic-Befehl hinzufügen : (return_value = <string>)

Hier eine Schritt für Schritt Anleitung um einen neuen Basic-Befehl hinzuzufügen :

Als Beispiel soll der Befehl „CHR$“ hinzugefügt werden
Der Befehl liefert den Character eines Ascii-Werts
Syntax : CHR$(<int_expr>) Return_Wert : <str>

1. File : „tokenizer.h“ ein neues Token am Ende der enum Liste hinzufügen
   Code : „TOKENIZER_CHRSTR“
2. File : „tokenizer.c“ das neue Token in der Keywordliste hinzufügen
   Code : {"CHR$", TOKENIZER_CHRSTR},
3. File : „tokenizer.c“ das neue Token in der Funktion „tokenizer_set_expression“ hinzufügen und zwar bei denen mit dem Rückgabewert <string>,
   Code : case TOKENIZER_CHRSTR :
4. File : „ubasic.c“ einen neuen Funktionsprototyp hinzufügen
   Code : static char *cmd_chrstr(void);
5. File : „ubasic.c“ die neue Funktion in der Funktion „*strpart(void)“ als neuen 'Case' hinzufügen
   Code :
```
case TOKENIZER_RIGHTSTR : // 'CHR$'
        r=cmd_chrstr();
break;
```
6. File : ubasic.c“ unter dem Abschnitt „NEW <STRING> STATEMENTS BY UB“ die neue Funktion ausprogrammieren
   Code :

```
static char *cmd_chrstr(void)
{
 char *ret_wert;

 return ret_wert;
};
```

FERTIG

Neuer Basic-Befehl hinzufügen : (return_value = <void>)

Hier eine Schritt für Schritt Anleitung um einen neuen Basic-Befehl hinzuzufügen :

Als Beispiel soll der Befehl „DELAY“ hinzugefügt werden
Der Befehl fügt eine kleine Pause ein
Syntax : DELAY(<int_expr>) Return_Wert : <void>

1. File : „tokenizer.h“ ein neues Token am Ende der enum Liste hinzufügen
   Code : „TOKENIZER_DELAY“
2. File : „tokenizer.c“ das neue Token in der Keywordliste hinzufügen
   Code : {"DELAY", TOKENIZER_DELAY},
3. File : ubasic.c“ unter dem Abschnitt „NEW <VOID> STATEMENTS BY UB“
   die neue Funktion ausprogrammieren
   Code :

```
static void delay_statement(void)
{
 int n;

 accept(TOKENIZER_DELAY);
 accept(TOKENIZER_LEFTPAREN);
 n=expr();
 accept(TOKENIZER_RIGHTPAREN);
}
```

1. File : „ubasic.c“ die neue Funktion in der Funktion „statement(void)“
   als neuen Case hinzufügen.
   Code :

```
 case TOKENIZER_DELAY
   delay_statement();
   break;
```

FERTIG