Funktionsframework für PicAxe M2 Typen

2017 by Matthias Heuschele / SSE

www.sse-web.de

Schwierigkeitsgrad 4 von 5

Vorwort

Das PicAxe Basic für die PicAxe M2 bietet keine anwenderspezifischen Funktionen, welche in fast 99% aller Programmiersprachen seit Jahren Anwendung finden.

Mit diesem Framework lassen sich rudimentäre Funktionskonstrukte im PicAxe Basic nachbilden, welche Argumente entgegennehmen und Funktionswerte zurückgeben können.

Dies ist mittlerweile der sechste Ansatz, ein einfaches, ressourcenschonendes und relativ schnelles Funktionsframework für das PicAxe Basic zu entwickeln. Alle Vorgänger-versionen hatten zumindest einen Makel: Mal war eine Version sparsam im Speicherverbrauch, aber sehr langsam. In der nächsten Version traf dann genau das Gegenteil zu und wenn alles schnell und ressourcenschonend ablief, war die Anwendung des Frameworks nicht sehr Benutzerfreundlich.

Mit dieser 6. Version bin ich der Meinung einen Kompromiss in Sachen Geschwindigkeit, Speicherauslastung und Benutzerfreundlichkeit gefunden zu haben und hoffe dass das Framework von dem ein oder anderen eingesetzt wird.

Abschließend bleibt noch die rhetorische Frage nach dem warum, denn schließlich gibt es genügend µC, die sich in Sachen Programmierung nicht auf dem Stand von 1980 befinden.

Hauptmerkmale des Frameworks

Lokale Variable

Innerhalb einer Funktion sind alle Variablen lokal und können unabhängig vom Hauptprogramm oder anderen Funktionen frei verwendet werden.

Argumentübergabe

An jede Funktion können bis zu 26 Argumente übergeben werden. Die Argumente werden über definierte PicAxe-Variablen an eine Funktionen übergeben. Der Anwender kann selbst entscheiden, ob eine Argument eine Referenz oder Wertübergabe sein soll.

Funktionsrückgabewert

Ob eine Funktion einen Wert zurückgibt, kann der Anwender frei entscheiden. Der Rückgabetyp einer Funktion ist immer ein WORD.

Globaler Stack

Auf dem globalen Stack können funktionsübergreifend Variablen zwischengespeichert und wiederhergestellt werden. Der globale Stack ist als FIFO-Stack (FirstInFirstOut) implementiert.

Lokaler Stack

Jede Funktion verfügt über seinen eigenen lokalen Stack, auf dem alle in der Funktion verwendeten Variablen zwischengespeichert und am Funktionsende wiederhergestellt werden.Interne Recourcen

Dieses Framework wurde so konzipiert, dass möglichst wenige PicAxe-Ressourcen (Variablen und RAM) für die interne Verwendung benötigt werden.

Variablen

Dieses Framework verwendet folgende Variablen für die interne Verwaltung. Diese Variablen sollten Sie in Ihren eigenen Programmen nur verwenden, wenn Sie die Funktionsweise dieses Frameworks genau verstanden haben.

Variable	Symbolname	Interne Verwendung
bptr		Lokaler Stackpointer
s_w1	GlobalStackPtr	Globaler Stackpointer
s_w2	Result	Funktionsrückgabewert
b26	LastVar	Interne Variable „LastVar“
b27	VarCounter	Interne Variable „VarCounter“

Tabelle 1

RAM

Dieses Framework belegt folgende Speicherbereiche für die interne Verwaltung:

Adresse	Verwendung	Größe in Byte
0-25	Freie Anwendervariable	26
26	Interne Variable „LastVar“	1
27	Interne Variable „VarCounter“	1
28-57	Lokaler Stack	30
58-77	Globaler Stack	20
78-511	Frei	434

Tabelle 2

Die Adressen und Größen der beiden Stacks sind als Defaultwerte zu betrachten und können je nach Anwenderbedürfnis geändert werden.

Stacks

Das Framwork beinhaltet zwei Stacks; den globalen Stack und den lokalen Stack. Der globale Stack kann vom Programmierer verwendet werden, um eine oder mehrere Variablen zwischenzuspeichern. Der globale Stack kann sowohl im Hauptprogramm als auch innerhalb von Funktion verwendet werden.

Der lokale Stack erfüllt prinzipiell den selben Zweck, unterscheidet sich aber in der Anwendung und sollte nur innerhalb von Funktionen verwendet werden.

Wichtige Programmierregeln

Bevor ich die Funktionsweise des Frameworks näher erläutere, vorab schon einmal die wichtigsten Regeln, die unbedingt beachtet werden sollten:

Jede innerhalb einer Funktion verwendete Variable, zu denen auch die Variablen für die Argumente gehören, muss vor ihrer Verwendung entweder auf dem lokalen Stack oder auf dem globalen Stack gesichert werden.
Vor dem beenden der Funktion müssen alle gesicherten Variablen wiederhergestellt werden. Zur Sicherung und Wiederherstellung dieser lokalen Variablen wird empfohlen, den lokalen Stack zu verwenden.

Alle in einer Funktion verwendeten Variablen, die unter Verwendung des lokalen Stacks gesichert und wiederhergestellt werden, sollten nach Möglichkeit ohne Zwischenraum aufeinander folgen und mit der ersten Variablen (`b0`) beginnen. Wenn eine Funktion z.B. fünf Bytevariable benötigt, sollten die Variablen `b0` bis `b4` verwendet werden.

Beispiel

Im folgenden Beispiel wird eine Funktion aufgerufen, die aus 3 übergebenen Argumenten den Mittelwert bildet und diesen zurückgibt. Auf die Details werde ich im nachfolgenden Abschnitt genauer eingehen.

```
001   ; *****************************************************************
002   ; Vorprogramm
003   ; *****************************************************************
004
005   ; -----------------------------------------------------------------
006   ; Interne Variablen deklarieren
007   ; -----------------------------------------------------------------
008   symbol GlobalStackPtr = s_w1
009   symbol Result = s_w2
010   symbol LastVar = b26
011   symbol VarCounter = b27
012
013   ; -----------------------------------------------------------------
014   ; Stackpointer initialisieren
015   ; -----------------------------------------------------------------
016   GlobalStackPtr = 56
017   bptr = 28
018
019   ; -----------------------------------------------------------------
020   ; Push und Pop Makros (Inline-Code)
021   ; -----------------------------------------------------------------
022   #DEFINE PushByte(ByteArg) poke GlobalStackPtr, ByteArg : inc GlobalStackPtr
023   #DEFINE PopByte(ByteArg) dec GlobalStackPtr : peek GlobalStackPtr, ByteArg
024   #DEFINE PushWord(WordArg) poke GlobalStackPtr, word WordArg : _
025         GlobalStackPtr = GlobalStackPtr + 2
026   #DEFINE PopWord(WordArg) GlobalStackPtr = GlobalStackPtr - 2 : _
027         peek GlobalStackPtr, word WordArg
028
029   ; -----------------------------------------------------------------
030   ; Funktionsprototypen
031   ; -----------------------------------------------------------------
032   ; Result = Mittelwert (b0 Arg1, b1 Arg2, b2 Arg3)
033
034
035
036   ; *****************************************************************
037   ; Hauptprogramm
038   ; *****************************************************************
```

```
039   Main:
040
041   b0 = 23 : b1 = 11 : b2 = 255 : call Mittelwert : w0 = Result
042
043   end
044
045
046
047   ; ********************************************************************
048   ; Funktionen
049   ; ********************************************************************
050
051
052   ; --------------------------------------------------------------------
053   ; Name:             Mittelwert
054   ; Prototyp:         Result = Mittelwert (b0 Arg1, b1 Arg2, b2 Arg3)
055   ; Funktion:         Ermittelt aus den uebergebenen Argumenten den Mittelwert
056   ;                   und gibt diesen zurueck
057   ; Variablen:        b0, b1, b2, b3 (w0 & w1)
058   ; --------------------------------------------------------------------
059   Mittelwert:
060
061         ;Variablen sichern
062         LastVar = 3 : Call PushVars
063
064         w1 = w1 + b0 + b1
065         Result = w1 / 3
066
067         ;Variablen wiederherstellen
068         LastVar = 3 : Call PopVars
069
070   return
071
072
073   ; --------------------------------------------------------------------
074   ; Name:             PushVars
075   ; Prototyp:
076   ; Funktion:         Interne Sub zur Verwaltung des lokalen Stacks
077   ; Variablen:
078   ; --------------------------------------------------------------------
079   PushVars:
080
081   for VarCounter = 0 to LastVar
082         Peek VarCounter, @bptrinc
083   next
084
085   return
086
087   ; --------------------------------------------------------------------
088   ; Name:             PopVars
089   ; Prototyp:
090   ; Funktion:         Interne Sub zur Verwaltung des lokalen Stacks
091   ; Variablen:
092   ; --------------------------------------------------------------------
093   PopVars:
094
095   dec bptr
096   do
097         Poke LastVar, @bptrdec
098         if LastVar = 0 then exit
099         dec LastVar
100   loop
101   inc bptr
102
103   Return
```

Vorprogramm: Zeile 1 - 35

Im Vorprogramm werden die Symbolnamen der für das Framework benötigten Variablen festgelegt (Zeile 5-12) und entsprechend initialisiert (Zeile 14-18). Die Variable `GlobalStackPointer` (`s_w1`) enthält die RAM-Adresse 58, die den Beginn des globalen Stacks kennzeichnet. Die Variable `bptr` enthält die RAM-Adresse 28, die den Beginn des lokalen Stacks kennzeichnet.

Die nachfolgenden Inline-Makros (Zeile 19-28) werden für die Verwendung des globalen Stacks benötigt. Die Anwendung und Funktionen der Inline-Makros und des globalen Stacks wird im Abschnitt Globaler Stack detailliert beschrieben.

In Zeile 32 erfolgt eine Kommentarzeile, die den Aufbau und die Argumente der verwendeten Funktionen beschreibt. Ich empfehle für jede Funktion eine solche Kommentarzeile anzulegen.

Hauptprogramm: Zeile 36 - 46

Im ersten Teil der Zeile werden den Argumenten der Funktion die entsprechenden Werte zugewiesen. Anschließend wird die Funktion (Sub) mit `call Mittelwert` aufgerufen. Nach beenden der Funktion (Sub) enthält die Variable `Result` das Ergebnis, welches in die Variable `w0` kopiert wird.

Funktionen: Zeile 47 - 103

Im Programmabschnitt Funktionen befinden sich Unterprogramme (Subs) welche die eigentlichen Funktionen darstellen.

Die Zeilen 52-72 enthalten die Funktion `Mittelwert`, die im Hauptprogramm mit `call Mittelwert` aufgerufen wird.

In Zeile 62 werden alle in der Funktion verwendeten Variablen vor ihrer Verwendung auf dem lokalen Stack gesichert bzw. gePUSHt

In Zeile 64-65 wird der Mittelwert aus den übergebenen Argumenten berechnet und der Variablen `Result` zugewiesen.

In Zeile 68 werden alle in der Funktion verwendeten Variablen, die in Zeile 62 auf dem lokalen Stack gepusht wurden, wiederhergestellt, d.h. vom lokalen Stack gePOPt.

In Zeile 70 wird wieder zurück ins Hauptprogramm gesprungen.

Interne Funktionen: Zeile 73 - 103

Die Subs `PushVars` und `PopVars` werden vom Framework benötigt, um die lokalen Funktionsvariablen (Variablen die innerhalb einer Funktion verwendet werden) auf dem lokalen Stack zu speichern bzw. vom lokalen Stack wieder zu lesen. Die Funktionsweise des lokalen Stacks soll nun im Detail erläutert werden.

Lokaler Stack

Sicherung

Wie bereits im Abschnitt Lokaler Stack beschrieben, müssen alle lokalen Variablen, zu denen auch die Argumente zählen, vor ihrer Verwendung gesichert werden. Wenn dazu wie im Beispiel der lokale Stack verwendet werden soll, geschieht dies mit folgender Zeile:

```
LastVar = 3 : Call PushVars
```

Mit diesem Code werden die ersten vier! Bytevariablen `b0`, `b1`, `b2` und `b3` auf dem lokalen Stack gesichert. Die Variable `LastVar` gibt an, welche die letzte zu sichernde Bytevariable sein soll. Wenn `LastVar = 3` ist, werden also alle Variablen, beginnend von `b0` bis `b3` auf dem lokalen Stack gesichert. Wenn `LastVar = 10` wäre, würden die Bytevariablen `b0` bis `b10` auf dem lokalen Stack gespeichert.

Um nicht Unnötig viele Variablen sichern zu müssen, empfiehlt es sich alle lokale Variablen aufeinander ohne Zwischenraum folgen zu lassen. Wenn z.B. eine Funktion nur zwei Variablen verwendet, sollten in der Funktion nur die Variablen b0 und b1 verwendet werden, die dann mit `LastVar = 1 : call PushVars` gesichert werden können. Prinzipiell könnte man z.B. auch die Variablen `b3` und `b15` verwenden, allerdings müsste man dann 16 Variablen (`b0` - `b15`) sichern und natürlich auch wiederherstellen, was die Performance nicht gerade erhöhen würde.

Wiederherstellung

Alle lokale Variablen, die auf dem lokalen Stack mit `call PushVars` gesichert wurden, müssen vor dem verlassen der Funktion mit

```
LastVar = X : Call PopVars
```

wiederhergestellt werden, wobei die Variable `LastVar` den gleichen Wert wie bei der Variablensicherung haben muss.

Globaler Stack

Der globale Stack dient in erster Linie dazu, Variablen vorübergehend zwischenzuspeichern und zu einem späteren Zeitpunkt wiederherzustellen. Der globale Stack ist als FIFO Stack implementiert, d.h. die zuletzt auf dem Stack abgelegte Variable wird als erste wieder vom Stack gelesen.

Der Zugriff auf den globalen Stack erfolgt mit vier Inline-Makros:

```
#DEFINE PushByte(ByteArg) poke GlobalStackPtr, ByteArg : inc GlobalStackPtr

#DEFINE PopByte(ByteArg) dec GlobalStackPtr : peek GlobalStackPtr, ByteArg

#DEFINE PushWord(WordArg) poke GlobalStackPtr, word WordArg : _
      GlobalStackPtr = GlobalStackPtr + 2

#DEFINE PopWord(WordArg) GlobalStackPtr = GlobalStackPtr - 2 : _
      peek GlobalStackPtr, word WordArg
```

Im Programm werden mit den folgenden Kommandos die Inline-Makros an die entsprechende Stellen im Programm eingefügt. Über diese Kommandos können dann Variablen auf dem globalen Stack abgelegt und wieder gelesen werden:

PushByte(b0) Pushbyte(123)	Ein Bytevariable oder Bytekonstante auf dem globalen Stack speichern

`PopByte(b1)`	`Ein Byte vom globalen Stack holen und in der angegebenen Variablen speichern`
`PushWord(w0)` `PushWord(9999)`	`Eine Wordvariable oder Wordkonstante auf dem globalen Stack speichern`
`PopWord(w3)`	`Ein Word vom globalen Stack holen und in der angegebenen Variablen speichern`

Es ist zu beachten dass zu jedem Push-Kommando ein korrespondierendes Pop-Kommando vorhanden sein muss.

Der globale Stack kann auch dazu verwendet werden, lokale Variablen innerhalb einer Funktion zu sichern und wiederherzustellen, obwohl sich dieses aufgrund der wachsenden Programmgröße nur empfiehlt, wenn sehr wenige lokale Variablen verwendet werden Wenn zur Sicherung und Wiederherstellung der globale Stack verwendet wird, müssen die lokalen Variablen im Gegensatz zum lokalen Stack nicht mehr aufeinander und ohne Zwischenraum folgen, da die Variablen in beliebiger Reihenfolge auf dem globalen Stack abgelegt werden können. Hierbei ist zu beachten, dass die Variablen vor dem Verlassen der Funktion nach dem FIFO Prinzip wieder hergestellt werden müssen.

Referenzargumente

In den bisherigen Ausführungen wurde immer davon ausgegangen, dass die lokalen Variablen, insbesondere die Argumente am Funktionsanfang auf dem Stack gesichert und am Funktionsende wieder vom Stack gelesen werden. Durch diese Vorgehensweise enthalten die Argumentvariablen denselben Inhalt, den sie auch beim Funktionsaufruf hatten. Innerhalb einer Funktion wird also keine Argumentvariable verändert, sondern nur der Wert der Variable an die Funktion übergeben. Dieser Vorgang wird auch als „Call by Value" bezeichnet.

In machen Fällen ist es aber ausdrücklich erwünscht, dass eine oder mehrere Argumentvariablen nach dem beenden einer Funktion einen neuen Wert beinhalten. Folgende Funktion, deren Prototyp hier dargestellt ist, setzt z.B. in der Argumentvariable `Wert` das Bit, das in der Argumentvariable `BitNr` angegeben wurde.

```
Result = BitSet (b0 Wert, b1 BitNr)
```

Die entsprechende Funktion könnte dann folgendermaßen lauten:

```
SetBit:

      ;Variablen (b0, b1, b2, b3, b4) sichern
      LastVar = 4 : Call PushVars

      ;Hier folgt der Algorithmus um das entsprechende Bit in b0 zu setzen
      ;Anschließend dann das Ergebnis (b0) auf dem globalen Stack ablegen
      push(b0)

      ;Variablen wiederherstellen
      LastVar = 4 : Call PopVars

      ;b0 (Ergebnis) wieder vom globalen Stack holen
      pop(b0)

return
```

In dieser Funktion werden mit Ausnahme der Argumentvariable `b0 Wert` keine Variablen verändert. Ein Rückgabewert `Result` ist nicht zwingen notwendig, könnte aber beispielsweise `true` oder `false` im Fehlerfall `BitNr > 7` zurückgeben.

Beispiele

Im folgenden einige Beispiele, die die Anwendung des Frameworks verdeutlichen sollen. Das Vorprogramm sowie die beiden Subs `PushVars` und `PopVars` entsprechen dem obigen Code, so dass ich diese in den Beispielen vorenthalte.

Beispiel 1

Die Funktion `SHL` (ShiftLeft) schiebt ein Byte um eine bestimmte Anzahl Bits nach links. Das Argument `Value` ist ein Referenzargument. Der Rückgabewert `Result` wird nicht benötigt.

```
01    ; ********************************************************************
02    ; Hauptprogramm
03    ; ********************************************************************
04    Main:
05
06    b0 = 8 : b1 = 2 : call SHL
07
08    ;b0 ist nach dem Aufruf 32 (2 x SHL)
09
10    end
11
12
13
14    ; ********************************************************************
15    ; Funktionen
16    ; ********************************************************************
17
18
19    ; --------------------------------------------------------------------
20    ; Name:             SHL (ShiftLeft)
21    ; Prototyp:         Result = SHL (b0 Value, b1 Count)
22    ; Funktion:         Schiebt die Bits in "Value" um die Anzahl "Count" nach
                          links.
23    ; Variablen:        b0, b1, b2
24    ; --------------------------------------------------------------------
25    SHL:
26          ;b2 Sichern, da dieses verändert wird
27          ;b0 muss nicht gesichert werden, da Referenzargument
28          ;b1 muss nicht gesichert werden, da diese nicht verändert wird
29          PushByte(b2)
30          b2 = 0
31          do
32                if b2 = b1 then exit
33                b0 = b0 * 2
34                b2 = b2 + 1
35          loop
36          PopByte(b2)
37    return
```

Beispiel 2

Die Funktion `SHR` (ShiftRight) schiebt ein Byte um eine bestimmte Anzahl Bits nach rechts. Das Argument `Value` ist ein Referenzargument. Der Rückgabewert `Result` wird nicht benötigt.

```
01    ; ********************************************************************
02    ; Hauptprogramm
03    ; ********************************************************************
04    Main:
05
06    b0 = 16 : b1 = 2 : call SHR
07
08    ;b0 ist nach dem Aufruf 4 (2 x SHR)
09
10    end
11
12
13
14    ; ********************************************************************
15    ; Funktionen
16    ; ********************************************************************
17
18
19    ; --------------------------------------------------------------------
20    ; Name:             SHR (ShiftRight)
21    ; Prototyp:         Result = SHR (b0 Value, b1 Count)
22    ; Funktion:         Schiebt die Bits in "Value" um die Anzahl "Count" nach
                          rechts.
23    ; Variablen:        b0, b1, b2
24    ; --------------------------------------------------------------------
25    SHR:
26          ;b2 Sichern, da dieses verändert wird
27          ;b0 muss nicht gesichert werden, da Referenzargument
28          ;b1 muss nicht gesichert werden, da diese nicht verändert wird
29          PushByte(b2)
30          b2 = 0
31          do
32                if b2 = b1 then exit
33                b0 = b0 / 2
34                b2 = b2 + 1
35          loop
36          PopByte(b2)
37    return
```

Beispiel 3

Die Funktion `SPI_ReadByte` liest ein Byte über die SPI-Schnittstelle ein und speichert dieses in `Result`.

```
01    ; ********************************************************************
02    ; Hauptprogramm
03    ; ********************************************************************
04    Main:
05
06    call SPI_ReadByte
07
08
09    end
10
11
12
13    ; ********************************************************************
14    ; Funktionen
15    ; ********************************************************************
16
17
18    ; --------------------------------------------------------------------
19    ; Name:             SPI_ReadByte
20    ; Prototyp:         Result = SPI_ReadByte
21    ; Funktion:         Liest ein Byte über SPI ein
```

```
22    ; Variablen: b0
23    ; ------------------------------------------------------------------
24    SPI_ReadByte:
25
26          ;Symbol SPI_IN = PinC.3
27          ;symbol SCLK = C.4
28
29          LastVar = 0 : Call PushVars
30
31          Result = 0
32
33          for b0 = 1 to 8
34                Result = Result * 2
35                if SPI_IN = 1 then
36                      Result = Result or 1
37                end if
38                pulsout SCLK,1
39          next b0
40
41          LastVar = 0 : Call PopVars
42
43    return
```

Beispiel 4

Die Funktion `SPI_ReadBytes` liest eine Anzahl Bytes über die SPI-Schnittstelle ein und speichert diese im RAM ab der angegebenen Adresse. Die Anzahl der zu lesenden Bytes muss größer 0 sein. Bei erfolgreicher Ausführung wird 1 zurückgegeben, ansonsten 0. Die Funktion `SPI_ReadBytes` ruft für jedes zu lesende Byte die Funktion `SPI_ReadByte` auf.

```
01    ; ********************************************************************
02    ; Hauptprogramm
03    ; ********************************************************************
04    Main:
05
06    b0 = 200 : b1 = 4 : call SPI_ReadBytes
07
08
09    end
10
11
12
13    ; ********************************************************************
14    ; Funktionen
15    ; ********************************************************************
16
17
18    ; ------------------------------------------------------------------
19    ; Name:             SPI_ReadBytes
20    ; Prototyp:         Result = SPI_ReadBytes (b0 Adress, b1 Count)
21    ; Funktion:         Liest eine Anzahl Bytes "Count" über SPI ein
22    ;                   und speichert diese im RAM ab Adresse "Adress"
23    ;                   "Count" muss > 0 sein.
24    ;                   Die Funktion gibt 1 zurueck wenn Ok, ansonsten 0
25    ; Variablen:        b0, b1, b2, b3
26    ; ------------------------------------------------------------------
27    SPI_ReadBytes:
28          if b1 > 0 then
29                LastVar = 3 : Call PushVars
30                b2 = b0 ; Adresse nach b2
31
32                for b3 = 1 to b1
33                      call SPI_ReadByte
34                      poke b2, Result
35                      inc b2
36                next b3
```

```
37                  Result = 1
38                  LastVar = 3 : Call PopVars
39            else
40                  Result = 0
41            end if
42      return
```