Kleines ASM Tutorial (1)

neuesthema antwort erstellen Parallax Propeller Forum Foren-Übersicht -> Assembler-Programmierung

Vorheriges Thema anzeigen :: Nächstes Thema anzeigen

Autor	Nachricht
deSilva Ass Anmeldungsdatum: 03.06.2007 Beiträge: 375	Verfasst am: 04.07.2007, 08:37 Titel: Kleines ASM Tutorial (1)

Propeller Assembler: Mittelstufe

Eigentlich braucht man doch nur das Manual zu lesen

Aber die Architektur des Propellers ist eben "innovativ" und nicht immer ist alles so, wie man sich das vielleicht „gedacht" hat. Hier deswegen ein kleiner Schnellkurs für Leute mit **schon einschlägigen Vorkenntnissen**

==1.
Außer einem 32k ROM Speicher, der uns hier weniger interessieren soll, besitzt der Propeller
- 32 kB RAM (8k 32-Bit Langworte)
- 8 Prozessoren („COGs") mit je 2 kB (512 32-Bit Langworte) Speicher
- einen 32 Bit I/O Port
- einen Taktzähler („CNT")
- 8 Semaphoren ("LOCKs")

==1b.
Nur zur Erinnerung: Beim RESET geschieht folgendes:
- Die ersten 32 kB eines an den Pins 28+29 angeschlossenen seriellen Speichers werden in den RAM gelesen
- Der etwa 2 kB große SPIN-Interpreter wird aus dem ROM-Speicher in den Prozessor #0 geladen und beginnt ab Speicheradresse 16 etwas zu interpretieren, was er für ein übersetztes SPIN-Programm hält.

== 2.
So, nun wieder zurück: Was kann so ein „Prozessor" überhaupt? Die immer richtige Antwort lautet: „Maschinenbefehle abarbeiten!". Ein üblicher Prozessor holt sich diese Befehle entweder aus einem gemeinsam adressierten Speicher („von-Neumann Architektur") oder aus einem ganz eigenem Speicherbereich („Harvard-Architektur"). Ein Propeller-Prozessor holt sich bei der „Initialisierung" 496 Langworte ab einer anzugebenden Adresse aus dem 32kB-RAM Speicher (oder auch aus dem ROM!) in seinen lokalen 2kB-Speicher. Nur diese Befehle können abgearbeitet werden! Aber: Nichts hindert den Programmierer daran, diese Befehle zu modifizieren (!) oder neue Befehle „nachzuladen"; Letzteres ist etwas trickreich, aber Ersteres absolut notwendig für das, was man üblicherweise „indizierte Adressierung" nennt.

Denn der lokale Speicherbereich des Prozessors wird universell auch als „Registerraum" verwendet! Das muss man sich mal vorstellen: Die Maschinenbefehle stehen in den „Registern", jeder in einem: Verrückt!

== 3.

Wir betrachten nun ein einfaches Beispiel. Wir können (siehe 1b.) allerdings nicht direkt ein Maschinensprachenprogramm starten, wie bei anderen Rechnern; wir müssen den „Umweg" über SPIN machen:

Code:

```
PUB Beispiel1
    cognew(@asm, 0)

DAT

asm     ORG  0
        MOV  dira,   #$FF     ' Setzte 255 in das Richtungsregister
        MOV  muster, #0       ' Löschen eines Registers
:rep    MOV  outa,   muster   ' Ausgeben des Bitmusters an P0 .. P7
        ADD  muster, #1       ' Inkrementiere das Registers
        JMP #:rep
muster LONG 0
```

"Defaultmäßig" läuft der Controller mit 12 MHz; bis auf Ausnahmen benötigt jeder Maschinenbefehl 4 Takte = 333 ns. Wir haben in der Schleife drei Befehle, bei jedem Durchlauf wird das LSB „getoggelt". Was wir jetzt also (hoffentlich) messen oder hören können, sind demnach Frequenzen von: 500, 250, 125, 62.5, 31.25, 15.6, 7.8 und 3.9 kHz an P0 bis P7. Ein Frequenzmesser wird aber bis zu 3% andere Werte anzeigen können.

Was sehen wir an dem Programm? ORG, MOV, ADD, JMP, $ für Hex, # für direkte (oder „immediate") Werte... Kennen wir doch alles! *DIRA* und *OUTA* sind Namen von Spezialregistern, die mit der I/O zutun haben: *DIRA/i]ist das Richtungsregister (1 = Ausgabe) und in [i]OUTA*wird dann die Ausgabe geschrieben; wenn im Richtungsregister das entsprechende Bit „an" ist, wird auch das I/O-Beinchen gesetzt.
MUSTER ist ein „freies" Register, eines das nicht durch Code belegt ist. Wir kennen in der Tat gar nicht die „Nummer" dieses Registers, wahrscheinlich wird sie 5 sein (Testfrage: Wieso?)

Was wir **nicht** sehen:
Alle Befehle sind gleich lang, nämlich genau 32 Bit, die sehr ordentlich aufgeteilt sind in Gruppen von jeweils
6 Bits: Befehlscode
3 Bits: Flag-Beeinflussung
1 Bit: Immediate-Adressierung
4 Bits: Ausführungsbedingung
9 Bits: Zielregisternummer
9 Bits: Quellregisternummer

Wir haben also einen 2-Adress-Befehlssatz vor uns, mit einer systematischen Direktadressierungsoption (0.. 511). Der Normalfall ist die absolute Adressierung von Registern mit ihrer „Nummer" (0..511). Eine andere Adressierung gibt es nicht. Wollen wir „indirekt" oder „indiziert" auf eines der vielen Register zugreifen, dann müssen wir den dazu verwendeten Befehl berechnen, bzw verändern. Hierfür gibt 2 ganz besondere Befehle MOVD und MOVS. Davon später mehr.

Ganz wichtig: Innerhalb der 2 k Byte des Prozessorspeichers gibt es **keine** Byteadressierung, sondern nur und ausschließlich 32-bit Registernummern.

== 3b.

Wir müssen doch noch einmal abschweifen. So ganz perfekt scheint das obige Programm nicht zu sein. Wir sind ja nicht die einzigen, die in diesem Controller herumwerkeln. O.k. wir haben uns vergewissert, dass wir P0 bis P7 benutzen dürfen, aber was ist mit P28, 29, 30, 31? Da war doch irgendwas.. Und wenn unser Zähler nach

ungefähr 20 Minuten in diesen Bereich kommt, dann geben wir da ja Bits aus, oder? Die Regel war doch etwa so:
- Ein Pin wird auf „Ausgabe" gesetzt, wenn wenigstens ein Prozessor im DIRA-Register das Bit gesetzt hat
- Ein Ausgabe-Pin wird auf „High" gesetzt, wenn wenigstens ein Prozessor in seinem OUTA-Register das entsprechende Bit gesetzt hat.
O.K. fast Wenn man das Prinzip-Schaltbild auf Seite 21 des Manuals gründlich studiert, dann findet man, dass der Inhalt des OUTA-Registers noch durch DIRA maskiert wird – wir müssen uns also weiter keine Sorgen machen!

== 4.
Schön, dass da ein Prozessor so vor sich hinläuft, dies sogar der Außenwelt durch Ohren beleidigende Rechteckschwingungen kund tut. Aber wie „kommunizieren" wir jetzt mit der „Innenwelt", dem 32k RAM-Speicher? Hierfür gibt es 6 Befehle, die den sonst üblichen LOAD- und STORE-Befehlen entsprechen:
- WRBYTE und RDBYTE
- WRWORD und RDWORD
- WRLONG und RDLONG

Doch wo bekommen wir die entsprechenden „Adressen" her? Eine Speicherzelle in einem Objekt hat ja keine „feste Adresse", sondern jede (wenn auch statische) „Instanz" benutzt eine neue. Wegen dieser Problematik wurde (in SPIN!) der @@-Operator geschaffen, den hoffentlich jeder kennt

Es gibt nun zwei Möglichkeiten:
(1) Bei der Initialisierung kann man einem Programm einen „Parameter" mitgeben (2. Wert im COGNEW, bzw. COGINIT); dieser Wert wird meistens als Anfangsadresse eines Speicherbereichs verstanden, ab dem die Information ausgetauscht wird. Diese Adresse findet das Assemblerprogramm dann im Register 496 (alias PAR) wieder. Achtung! Dies muss eine Long-Adresse sein, also durch 4 teilbar, weil sie während des Transfers auf 14 bit gekürzt und danach wieder expandiert wird!! Übergibt man etwas anderes, dann darf man sich nicht wundern, wenn die letzten beiden Bits zu Null geworden sind.
(2) Die zweite Möglichkeit ist etwas trickreicher, aber durchaus im Rahmen der Philosophie des Propellers. Das (übersetzte) Assemblerprogramm des DAT-Bereiches steht ja im RAM und kann – bevor es in den Prozessor geladen wird – problemlos durch das SPIN-Programm modifiziert werden

Code:

```
VAR
    LONG zaehler

PUB Beispiel2
    Ziel := @zaehler
    cognew(@asm, 0)

DAT
asm     ORG     0
        MOV     dira,   #$FF        ' Setzte 255 in das Richtungsregister
        MOV     muster, #0          ' Löschen eines Registers
:rep    MOV     outa,   muster      ' Ausgeben des Bitmusters an P0..P7
        ADD     muster, #1          ' Inkrementiere das Registers
        WRLONG  muster, ziel
        JMP     #:rep
muster  LONG    0
ziel    LONG    0                   ' Hier wird die Zieladresse hinterlegt
```

oder

Code:

```
VAR
    LONG zaehler

PUB Beispiel2b
    cognew(@asm, @zaehler)

DAT
asm     ORG     0
        MOV     dira,   #$FF    ` Setzte 255 in das Richtungsregister
        MOV     muster, #0      ` Löschen eines Registers
:rep    MOV     outa,   muster  ` Ausgeben des Bitmusters an P0..P7
        ADD     muster, #1      ` Inkrementiere das Registers
        WRLONG muster, par
        JMP     #:rep
muster  LONG    0
```

Wir bemerken hier:
- Die WR...-Befehle arbeiten „links nach rechts" (während sonst MOV und RD... „von rechts nach links" transportiert. Allso aufgepasst: Das ist eine böse „Falle"!
- Unser „Timing" ist kaputt gegangen; WRLONG braucht nicht 4 Takte sondern zwischen 7 und 22.

== 5.

Innerhalb des Assemblerprogramms kann man auch lokale Unterprogramme aufrufen (natürlich keine Objekt-Methoden!). Das geht nicht so einfach wie man das aus anderen Prozessorarchitekturen kennt, bei denen ein CALL-Befehl die „Rücksprungadresse" in einen Stack legt, wo sie ein RET-Befehl wieder raus nimmt. CALL und RET sind bekanntlich auch nicht die schnellsten Befehle.... Bei Propeller aber geht alles schnell nämlich in 4 Takten: Die Rücksprungadresse wird in ein Register geschrieben, und man muss selbst aufpassen, dass man dann hierhin zurückkehrt.

Code:

```
` Beispiel 3
DAT
asm     ORG     0
        MOV     m10par, #30         ` diese Zahl 30 ...
        JMPRET mal10_ret, #mal10    ` ... soll mit 10 multipliziert
werden

        .....                       ` hier geht's dann irgendwie weiter

mal10     MOV     m10par2, m10par   ` eine Kopie wird gemacht
          SHL     m10par2, #2       ` das ist mal 4
          ADD     m10par, m10par2   ` plus 1 = 5
          ADD     m10par, m10par    ` mal 2 = 10
          JMP     mal10_ret         ` indirekter Sprung
mal10_ret  LONG   0
m10par     LONG   0
m10par2    LONG   0
```

Wir bemerken Folgendes :
- Das ist BEI WEITEM nicht so bequem wie ein Hardware-unterstützter Stack mit PUSH und POP (oder auch auto-inkrementierender indizierter Adressierung); o.k. das ist ein bisschen der Unterschied zwischen RISC und CISC J
- Wir könnten eventuell mit einem einzigen „Rücksprungregister" auskommen, wenn wir nur eine Prozedurverschachtelng haben; „Rekursion" ist sowieso außerhalb jeder Diskussion hier.
- Wenn wir in der Routine allerdings nur einen einzigen Rückkehrpunkt haben, dann könnten wir die Rücksprungadresse auch gleich in den (Rück-)Sprungbefehl einkopieren. Voraussetzung wäre allerdings, dass der JMPRET-Befehl nicht das gesamte Register beschreibt, sondern nur die untersten 9 Bits – und das genau tut der JMPRET-Befehl auch nur!!

Wir ändern also ab:

Code:

```
` Beispiel 3b
mal10     MOV    m10par2, m10par   ` eine Kopie wird gemacht
          SHL    m10par2, #2       ` das ist mal 4
          ADD    m10par, m10par2   ` plus 1 = 5
          ADD    m10par, m10par    ' mal 2 = 10
mal10_ret  JMP    #0                ` direkter Sprung auf noch
unbekanntes Ziel
m10par     LONG   0
```

So, jetzt ist noch lästig, dass der Aufrufer wissen muss, wo der Rücksprungbefehl liegt.

JMPRET mal10_ret, #mal10

sieht ja an sich auch schon ein bisschen verrückt aus. Deshalb gibt es folgende Assembler-Konvention:

Code:

```
Statt
   JMPRET mal0_ret, #mal10
kann (und sollte !) man auch
     CALL #mal10
schreiben, und statt
      Mal10_ret  JMP    #0
      Mal10_ret  RET
```

Das sind **keine** Maschinenbefehle, sonder nur Assembler-Abkürzungen zur Verbesserung der Lesbarkeit!

Code:

```
` Beispiel 3c
DAT
asm       ORG    0
          MOV    m10par, #30       ` diese Zahl 30 ...
          CALL   #mal10            ` ... soll mit 10 multipliziert werden

          .....                     ` hier geht's dann irgendwie weiter

mal10     MOV    m10par2, m10par   ` eine Kopie wird gemacht
          SHL    m10par2, #2       ` das ist mal 4
          ADD    m10par, m10par2   ` plus 1 = 5
          ADD    m10par, m10par    ' mal 2 = 10
mal10_ret RET
m10par     LONG   0
m10par2    LONG   0
```

Zuletzt bearbeitet von deSilva am 04.07.2007, 08:42, insgesamt einmal bearbeitet

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 08:39 Titel:

== 6. Bedingte Befehle

Der nützlichste Befehl ist vielleicht der DJNZ-Befehl, den es in nahezu derselben Form auf den meisten Prozessoren gibt. Ein Beispiel folgt später – er funktioniert genau so wie man denkt.

Alle anderen Bedingungen hängen an „Flags": Z wie Zero, C wie Carry. Manche Prozessoren haben bis zu 8 Flags, oder sogar noch mehr. Solche „Flags" werden üblicherweise bei gewissen Befehlen „automatisch" gesetzt, als Nebeneffekt oder auch allein in der Absicht, ein Flag zu setzen. Benutzt werden diese „Flags" als Bedingung für

einen Sprung- oder Skipbefehl. Wie sollte es anders sein, ist dies auch beim Propeller etwas anders.

- Es gibt genau zwei Flags Z und C
- Jeder (!) Befehl kann in Abhängigkeit von diesen Flags ignoriert werden; es verhält sich also so, als ob vor jedem Befehl ein bedingter SKIP-Befehl stehen würde.
- Wie viele Bedingungen gibt es nun? Es ist die gleiche Frage wie: Wie viele logische Verknüpfungen gibt es bei 2 Eingängen? Die Antwort ist: 16 und alle sind implementiert! Man hat sich Mühe gegeben, jedem von ihnen einen „sinvollen" Namen zu geben, deshalb gibt es auch einige Doppelbenennenungen: IF_C, IF_E, IF_Z, IF_C_OR_Z u.s.w. u.s.f.
- Dieses „Präfix" kann vor jedem Befehl stehen, davon wird er nicht länger oder dauert länger, denn diese Bedingung ist schon in den 32-Bits des Befehls als 4-Bit-Feld eingebaut. Es dauert einige Zeit, bis man die sich dadurch auftuenden Möglichkeiten rafft.

Die andere Frage ist: "Wie werden denn nun diese beiden Flags gesetzt?" Eine gute Frage, denn jeder, der schon mal Assembler programmiert hat, weiß, das man sich das nur schwer merken kann. Beim Intel 8080/Z80, den ich mal sehr intensiv programmiert habe, gibt es einen Befehl INR (der 8-Bit-Register inkrementiert), der Flags setzt, und einen Befehl INX (der 16-Bit Adressregister inkrementiert), der das nicht tut. Das ist sehr praktisch, nahezu lebensnotwendig!

Beim Propeller wird nun bei fast allen Befehlen ein die "Flag.Wechsel" angeboten, aber nur dann realisiert, wenn man **hinter** den Befehl ein Postfix **WZ** oder **WC** setzt. Auf Grund welches Sachverhalts sich dieses Flag ergibt, muss im Manual studiert werden! Es gibt übliche (Z = Register ist Null oder Leer, C = arithmetischer Überlauf, d.h. Wechsel von Plus auf Minus oder umgekehrt,...) und unübliche Verwendungen (C = Parität des Registers! So wird es bei MOV gesetzt. Ja, auch MOV **kann** die Flags setzen, was bei anderen Assemblersprachen eher ein "no-go" wäre. Ein "TEST" Befehl ist deswegen völlig unnötig. Es gibt allrdings ein Assembler-Makro, das TEST heißt und hinter dem sich der AND-Befehl versteckt. Wie das? AND würde doch das Zielregister VERÄNDERN??

Nun, eines der vielen Ungewöhnlichkeiten beim Propeller ist die Möglichkeit, das Zurückschreiben des Registers zu verhindern (Im Prinzip folgen ja die meisten Befehle dem Read-Modify Write-Muster.) Die wird durch das (zusätzliche!) Postfix **NR** angegeben. Zusätzlich deswegen, weil ein NR ohne ein WC oder WZ nun wirklich keinen Sinn macht - das wäre ja der NOP-Befehl. Mit dieser NR-Technik kann man nun nach Herzenslust Flags setzen, z.B. über den SHIFT-Befehl mit NR sich jedes beliebige Bit ins Z oder C Flag zu setzen. Man beachte: Auch ein 30-Bit-Shift Befehl dauert nur 4 Takte - ein echtes High-End-Feature!!

(Achtung in den beiden letzten Absätzen gab es anfangs ein Missverständnis - habe ich umgeschrieben 1.7.2007)

Nach soviel Theorie zwei Beispiele: Wir wollen zählen, wie viele Bits in einem 32 Wort gesetzt sind.

Code:

```
'Beispiel 4
dasWort   long $XXXXXXXXXX
zaehler    long 0
ergebnis   long 0
      MOV ergebnis, #0    ' hier entsteht das Ergebnis
      MOV zaehler,  #32   ' wir untersuchen jedes der 32 Bits
:loop ROL dasWort, #1 WC  ' Setzte Carry auf Bit 31 und rotiere links
      IF_C ADD ergebnis, #1
      DJNZ zaehler, #:loop
```

Dieses Programm hat mehrere Vorteile: *dasWort* steht am Ende unverändert in seinem

Register, es gibt eine exakte Zeit, die es dauert, und in der vorletzten Zeile könnte man statt ADD ergebnis,#1 auch etwas ganz anderes hinschreiben.

Ein Alternativprogramm sieht so aus:

Code:

```
'Beispiel 4b
dasWort    long $XXXXXXXXXX
ergebnis    long 0
      MOV ergebnis, #0   ` hier entsteht das Ergebnis
:loop SHR dasWort, #1 WC WZ ` Setzte Carry auf Bit 0 und shifte nach
rechts
                           ` Setze außerdem Z, wenn das Register leer
ist
      ADDX ergebnis,#0     ` Addiere mit Carry
      IF_NZ JMP #:loop
```

Diese Programm zerstört *dasWort*, ist aber in der Regel schneller fertig; das kann ein Vor- oder auch ein Nachteil sein.... Wir brauchen auch keinen Hilfszähler mehr. Warum shiften wir nach rechts? Nun, wir haben angenommen, dass *dasWort* in der Regel nur kleine Werte enthält, so dass es sich durch shiften nach rechts eventuell etwas schneller leert.

Zuletzt bearbeitet von deSilva am 04.07.2007, 09:10, insgesamt einmal bearbeitet

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 08:40 Titel:

== 7. Ungewöhnliche Befehle

Das ist ja die immer die erste Frage: Was kann „er" denn überhaupt? (Merke: Schiffe sind weiblich, Computer männlich)
(a) „Number-Crunchen" kann man vergessen; die Floatingpoint-Simulation ist langsam, aber immerhin ist eine vorhanden
(b) Einen Multiplikations- oder gar Divisionsbefehl sucht man vergebens
(c) 32-bit Addieren (ADD), Subtrahieren (SUB) und Vergleichen (CMP) von Vorzeichen losen und Vorzeichen behafteten (...S) Zahlen, mit Unterstützung für mehrfache genaue Arithmetik (...X)
(d) MOV, NEG, ABS und ABSNEG; man beachte, dass es sich hier um 2-Adress-Befehle handelt, also NEG und ABS kopieren auch in ein anderes Register; Ein Beispiel

Code:

```
NEG rega, #511    ` jetzt steht -511 in rega
```

(e) Logisch-bitweise Verknüpfungen (AND, ANDN, OR, XOR, TEST)
(f) Komplette Reihe von Shiftbefehlen, mit **beliebiger** Shiftgröße (0..31) direkt oder aus Register (RCL RCR ROL ROR SHR SHL SAR)

Jetzt kommen wir langsam zu den „ungewöhnlichen Befehlen", die es auf anderen Rechnern seltener gibt:

(g)
MAX a,clipval
Dieser Befehl müsste eigentlich MIN heißen :- vielleicht noch besser „Upper Clipping": Er setzt das Register a auf clipval, wenn es vorher größer gewesen war.
MIN a, clipval
„Lower Clipping": Setzt Register a auf clipval, wenn es vorher kleiner gewesen war. Also Vorsicht wegen der falschen Namen!
CMPSUB a,b
Subtrahiert b von a, aber nur falls a dadurch nicht negativ wird! Da es keinen Divisionsbefehl gibt kann man sich hiermit in manchen Situationen behelfen; Beispiel:

Code:

```
` Beispiel 7
` Berechne  c := a dividiert durch b; beides positive Zahlen
      MOV     c, #0
:loop  CMPSUB  a,  b wc wz `Carry wird gesetzt, wenn vorher b=>b
  IF_C   ADD   c, #1 nc nz
  IF_C_AND_NZ   JMP  #:loop
` Der Rest steht in a
```

Wir könnten auch schrieben (ohne weiteren Vorteil)

Code:

```
` Berechne c:=a/b; beides positive Zahlen
      NEG     c, #1
:loop    ADD  c,  #1
      CMPSUB  a,  b wc wz `Carry wird gesetzt, wenn vorher b=>b
  IF_C_AND_NZ   JMP  #:loop
` Der Rest steht in a
```

(h)
TJZ r, ziel
TJNZ r, ziel
Sprung, falls das Register r Null ist oder nicht Null
DJNZ r, ziel
Wie TJNZ, jedoch wird vorher noch das Register r dekrementiert
Auch diese Befehle dauern nur 4 Takte WENN sie springen; ist die Sprungbedingung nicht erfüllt, dann braucht der Prozessor weitere 4 Takte, um wieder die „normale" Abarbeitungsreihenfolge einzustellen

(i)
REV a, n
Löscht die obersten n Bits und kehrt die Reihenfolge der restlichen (unteren) 32-n Bits um

(k)
4 Befehle **MUX* r, maske**
Setzt alle Bits im Register r, die eine 1 in der Maske haben auf den Wert von C, NC, Z oder Z, je nach *
D.h. Abhängig von den Flags C, NC, Z oder NZ wird ein OR oder ein ANDN durchgeführt

4 Befehle **NEG* ziel, quelle**
Abhängig von den Flags C, NC, Z oder NZ wird ein MOV oder ein NEG durchgeführt

4 Befehle **SUM* summe, quelle**
Abhängig von den Flags C, NC, Z oder NZ wird ein ADDS oder ein SUBS durchgeführt

(l)
SUBABS a,b
Der Absolutbetrag von b wird von a abgezogen

(m)
CLKSET

COGID
COGINIT
COGSTOP

LOCKNEW
LOCKRET
LOCKCLR
LOCKSET
WAITCNT
Ähnlich wie die entsprechenden SPIN-Befehle

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 08:41 Titel:

Ich poste hier drei (offizielle Parallax-) Beispiele; im Moment englisch kommentiert, ich füge da gelegentlich meine Anmerkungen dazu.

Testfrage: Was ist der Unterschied zwischen dem 3. Beispiel unten (Division) und dem Beispiel 7 oben?

Code:

```
' Compute square-root of y[31..0] into x[15..0]
'
root            mov     a,#0            'reset accumulator
                mov     x,#0            'reset root
                mov     t,#16           'ready for 16 root bits

:loop           shl     y,#1    wc      'rotate top two bits of y into
accumulator
                rcl     a,#1
                shl     y,#1    wc
                rcl     a,#1
                shl     x,#2            'determine next bit of root
                or      x,#1
                cmpsub  a,x     wc
                shr     x,#2
                rcl     x,#1
                djnz    t,#:loop        'loop until done

root_ret        ret                     'square root in x[15..0]
```

Code:

```
'
' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
multiply    shl   x,#16     'get multiplicand into x[31..16]
            mov   t,#16     'ready for 16 multiplier bits
            shr   y,#1 wc   'get initial multiplier bit into c
:loop  if_c add   y,x wc    'if c set, add multiplicand into product
            rcr   y,#1 wc   'get next multiplier bit into c, shift
product
            djnz  t,#:loop 'loop until done

multiply_ret ret 'return with product in y[31..0]
```

Code:

```
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide      shl y,#15       'get divisor into y[30..15]
            mov t,#16       'ready for 16 quotient bits
:loop       cmpsub x,y wc 'if y =< x then subtract it, quotient bit
into c
            rcl x,#1        'rotate c into quotient, shift dividend
            djnz t,#:loop 'loop until done
divide_ret ret              'quotient in x[15..0], remainder in x[31..16]
```

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 09:15 Titel:

== 8. Indizierte Adressierung

Indizierte Adressierung wird benötigt, wenn wir aus einem Vektor ein berechnetes Element herausholen wollen: X[i]. „Indirekte" Adressierung ist hiervon ein Spezialfall: X[0]. Verschiedenen Prozessoren haben hier recht unterschiedliche Konzepte, manchmal eingeschränkt auf die Größe („8-Bit-Index"), manchmal sogar äußerst flexibel („Postincrement/Predecrement"). Der Propeller hat der Einfachheit wegen da gar nix

Wohlgemerkt, wenn wir auf den globalen RAM zugreifen (mit RDWORD u.s.w.), dann müssen wir sowieso eine Adresse berechnen, denn dieser Zugriff geschieht über ein Register (ist – technisch gesprochen – also „indirekt"). Wenn X also die Basis Adresse eines Vektors im globalen RAM ist und wir auf das I-te Langwort zugreifen wollen, dann sieht das so aus:

Code:

```
  MOV r, I
  SHL r,2       ` mal 4 = Byteadresse
  ADD r, X
  RDLONG r,r
```

Hier die Summe von 20 (Vorzeichen behafteten) Zahlen

Code:

```
  MOV adr, X
  MOV summe, #0
  MOV anzahl, #20
:loop
  RDLONG r, adr
  ADDS summe, r
  ADD adr, #4       ` nächstes long
  DJNZ anzahl, #:loop
```

Aber wie machen wir das, wenn wir die Elemente es Vektors im Prozessorspeicher haben, z.B. weil wir kompliziertere Matrizenoperationen durchführen wollen, ohne immer auf den RAM zu zu greifen?

Hier bleibt uns nichts weiter, als „selbstmodifizierenden" Code zu schreiben!

Code:

```
        ORG 0
        MOVS :zugrf, #X     ` in Register X bis X+19 stehen 20 Longs
        MOV summe, #0
        MOV anzahl, #20
:loop
:zugrf  ADDS summe, 0         ` die untersten 9 Bits dieses Befehls
werden verändert
         ADD :zugrf, #1       ` nächstes Register
         DJNZ anzahl, #:loop
.....
X:       RES 20
```

Der aufmerksame Leser hat's gemerkt: „MOVS – watt'n dat??". Diesen Befehl habe ich bisher unterschlagen: Es gibt drei MOV-Befehle, die nicht das GANZE Zielregister setzen, sondern nur jeweils 9 Bits daraus:
- die untersten 9 Bits (0..8) MOVS (S wie „source")
- die nächsten 9 Bits (9..17) MOVD (D wie „destination")

- die nächsten 9 Bits (9..17) MOVD (D wie „destination")
- die ersten 9 Bits (23..31) MOVI (I wie „instruction")

Die Menomotechnik bezieht sich auf den Aufbau eines Befehls. Aber auch einige I/O-Kontrollregister sind so organisiert, dass man mit diesen Befehlen und ohne „Masken" ganz bequem einige Felder setzen kann. Für die 5 Bits „dazwischen" (18.22) gibt es keinen besonderen Befehl.

Noch einige Anmerkungen:
- Die Schleife im unteren Beispiel läuft in 12 Takten, im oberen Beispiel in 24 Takten; falls aber statt einer 1-Befehls-Addition etwas Komplizierteres passiert, ist der Unterschied nicht mehr so groß.
- Es darf niemals der drauf folgende Befehl modifiziert werden! Das liegt wohl daran, dass der nöchste Befehl schon ausgelesen wird, bevor der letzte Befehl ausgeführt wurde („Instruction Prefetch")
- Es gibt Berichte, dass der ORG-Anweisung und die RES nicht ganz richtig funktionieren: ORG sollte im Moment immer bei „Null" beginnen, und Hinter "RES" - Anweisungen sollten keine anderen Anweisungen mehr stehen. Dies ist aber – wenn es denn wahr ist – kein Fehler des Propellers sondern der IDE und wird sicherlich in Kürze korrigiert.

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 10:01 Titel: ANmrkungen zu RES und ORG

Kaio hat dankenswerter Weise noch einige Erläuterungen zu ORG und RES abgegeben.
Die Regel ist:
- ORG "wirkt nicht "!
- Hinter RES keine Speicherbesetzungen mehr (LONG oder Befehle)!

Man kann dieses "Feature" leicht verstehen, wenn man sich die Ladestrategie der COGs vor Augen führt (das ist "Hardware"!)

Bei einem COGNEW (addr, p) geschieht ganz genau folgendes:
(1) 2 K des HUB Speichers ab der Langwort-Adresse addr werden von Anfang bis Ende in das (den?) nächste(n) freie(n) COG übertragen, ohne Rücksicht auf Verluste!

Aus Optimierungsgründen macht es aber keinen Sinn, im begrenzten HUB "Leerstellen" zu lassen; deshalb wird dort kein Platz reserviert für RES-Variablen. Dadurch bricht nach einer RES Variable die Zuordnung von HUB und COG Adressen völlig zusammen. Das gleiche würde bei ORG geschehen.

Die Abhilfe ist natürlich, im HUB die RES-Zellen und ORG Lücken mit Nullen auszupadden.... Das will aber ernsthaft auch keiner... So ist diese sehr ärgerliche Falle entstanden!

Und allergrößte Vorsicht bei der Benutzung von globalen DAT-Variablen von SPIN aus! Dies ist natürlich möglich und sinnvoll; auf diese Weise können COGs sehr einfach parametrisiert werden, weil diese Zellen ja erst zum Zeitpunkt des COGNEW-Befehls geladen werden. Danach besitzt der COG natürlich nur diese KOPIE bis zum nächsten COGNEW.

Aber: Niemals von SPIN aus eine DAT-Variable benutzen, die mit RES deklariert wurde!! Ist jetzt wohl jedem klar, warum nicht

(2) Der Assembler hat frecherweise bereits 16 Bits vorne und 2 Bits hinten von p abgeschnitten (p>>2 & $3fff) und diese Zahl (max 14 Bits) - wieder um 2 nach links geshiftet - wird ins PAR Register des COGs gespeichert.

Dies Letztere nur zur Vervollständigung (und weil hier ja auch eine kleine Falle lauert)

Üblicherweise wird COGNEW so aufgerufen:
COGNEW(@asm, @parblock)
Wenn man keine @ angibt, muss man genau wissen, was man tut

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.07.2007, 11:36 Titel:

=== 9. Locks und Semaphoren

Hier gibt es viel Unsicherheit! Brauche ich das? Wann? Das habe ich noch nie gebraucht!

Fakt ist, das jedes Parallel-System - egal, ob Hardware oder Software-simuliert - zwingend solche Locking-Mechanismus benötigt. So existieren sie auch im Propeller, und da natürlich "in Hardware".

Die Problemlage wollen wir an folgendem Beispiel studieren:
In einem Kaufhaus mit nur einem Eingang und einem Ausgang soll kontrolliert werden, wie viele Kunden (oder Angestellte) sich im Gebäude befinden, so kann das Gebäude abends schneller abgeschlossen werden und das Management bekommt stündliche Berichte und Statistiken über die Nachfrage und kann den Personaleinsatz besser steuern. Eine zuverlässige Lichtschranke am Eingang und Ausgang ist schnell installiert....

Es gibt jetzt zwei Möglichkeiten, die Frage zu beantworten: "Wie viele Menschen befinden sich im Gebäude":

(1)

Code:

```
wenn LichtschrankeEingang dann InCounter +=1
wenn LichtschrankeAusgang dann OutCounter +=1

Anwesende := InCounter - OutCounter
```

Dieser Ansatz hat mehrere Nachteile:
- die Counter können "überlaufen" obwohl kaum jemand im Gebäude ist
- Das Ergebnis muss immer "berechnet" werden und steht nicht "statisch" zur Verfügung

Wenn jedoch Beides kein Problem sein sollte, dann ist diese Lösung immer vorzuziehen!

(2)

Code:

```
wenn LichtschrankeEingang dann Anwesende +=1
wenn LichtschrankeAusgang dann Anwesende -=1
```

Das ist nahe liegend und simpel, doch hier versteckt sich eine Falle! Wird der Code - so wie angegeben - seriell abgearbeitet, dann muss nur noch sichergestellt werden, dass während der Bearbeitungszeit nicht etwa ZWEI gleiche Ereignisse eintreffen.

Wollen wir das durch getrennte Hardware oder (Software-) Parallelisierung ausschließen, so wird auf den Akkumulator "Anwesende" von zwei Stellen möglicherweise GLEICHZEITIG zugegriffen.

Klar gibt es sowas wie GLEICHZEITIG nicht (Dies ist übrigens ein spannendes Grundproblem der sogenannten "Digitalisierung": An IRGENDEINER Stelle muss in einem Digitalen System ja mal aus der analogen Umwelt gewandelt werden....)

Nun sehen wir uns aber mal den Code an: COGE lauert an der einen und COGA an der anderen Lichtschranke:

Code:

```
'' Kaufhaus 1
CON
aPin = 2
ePin = 3
VAR
long Anwesende

PUB  main
Anwesende := 0
cognew(@Eingang, @Anwesende)
cognew(@Ausgang, @Anwesende)

DAT
Eingang
  waitpeq :null, #ePin
  waitpne :null, #ePin
  rdlong :e, par
  add :e, #1
  wrlong :e, par
  jmp #Eingang

:null long 0
:e     res 1

Ausgang
  waitpeq :null, #aPin
  waitpne :null, #aPin
  rdlong :a, par
  sub :a, #1
  wrlong :a, par
  jmp #Eingang

:null long 0
:a     res 1
```

Hier wird deutlich, dass wir bei folgendem zeitlichen Ablauf in Schwierigkeiten kommen:

RDLONG :a
RDLONG :e
WRLONG :a
WRLONG :e

Ein eintretender Besucher wird nicht mitgezählt. Wie hoch die Wahrscheinlichkeit für diese Überlappung ist, wissen wir nicht.

Wie kann man das Problem PRINZIPIELL lösen? Nun, wir müssen aus den drei Befehlen RDLONG ADD WRLONG eine unaufbrechbare Einheit machen!

Die Lösungen in "simulierten" Parallelsystemen sind hier bekannt und sehr einfach:
- Setzen der Interruptsperre
- Besonderer Maschinenbefehl "ReadAndModify" (seltener)

Das hilft uns aber nicht weiter, wenn wir zwei echte GETRENNTE Prozessoren haben... Doch der eben angesprochene Befeht "ReadAndModify" - anwendet auf den HUB-Speicher - ist die Lösung.

Dieser Befehl heißt beim Propeller LOCKSET (bzw. LOCKCLEAR); allerdings wird kein HUB-Speicher benutzt, sondern nur 8 spezielle Bits.

Wir packen also unsere kritische Zähleroperation so ein:

Code:

```
:l  lockset sema  WC
```

```
    if_c jmp #:1    : warten, dass der Partner den "kritischen Bereich"
verlässt
  rdlong :a, par
  sub :a, #1
  wrlong :a, par
  lockclr sema
```

Ein geändertes Programm sieht dann so aus; wir haben noch einige Optimierungen durchgeführt, dessen Verständnis den interessierten Leser sicherlich weiterbringt

Code:

```
'' Kaufhaus 2
VAR
long Anwesende

PUB  main

Anwesende := 0
semaNummer := locknew
zaehlerAdresse := @Anwesende

pin := 2
delta := 1
cognew(@Waechter, 0)   ' Eingang
pin := 3
delta := -1
cognew(@Waechter, 0)   'Ausgang

DAT
Waechter
  waitpeq :null, pin
  waitpne :null, pin
:w  lockset semaNummer WC
  if_C jmp :w
  rdlong a, zaehlerAdresse
  add a, delta
  wrlong a, zaehlerAdresse
  lockclr semaNummer
' ggf. entprellen des Schalters
  mov a, :entprellzeit
  add a, cnt
  waitcnt a, #0
  jmp #Waechter

:null long 0
:entprellzeit  long 80_000_000* 10/1000   ' 10 ms
pin  long 0
delta long 0
semaNummer    long 0
zaehlerAdresse long 0
a   res 1
```

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 05.07.2007, 13:19 Titel:

O.k. Ich war so unvorsichtig den o.a. Code mal auszuprobieren - er funktionierte natürlich nicht

Warum?

Code:

```
pin := 2
delta := 1
cognew(@Waechter, 0)   ' Eingang
pin := 3
delta := -1
cognew(@Waechter, 0)   'Ausgang
```

Während die COGs geladen werden, läuft das Hauptprogramm natürlich weiter, d.h. die globalen SPeicherzellen pin und delta sind schon lange überschrieben, wenn sie zum Eingangs-COG übertragen werden. Parallelrechnen ist eben nicht so einfach....

Es gibt mehrere Möglichkeiten, das Problem zu lösen:
- Warteschleife hinter dem ersten COGNEW (2 k * 16 = 32 k Takte)
- "Ordentliche" Parametrisierung über PAR
- "Unordentliche" Parametrisierung über PAR

Ich gebe mal ein Beispiel für den letzten Fall:

Code:

```
PUB  main

Anwesende := 0
semaNummer := locknew
zaehlerAdresse := @Anwesende

'pin 2
cognew(@Waechter,  constant(2*4))  ' Eingang

' pin 3, decrement
cognew(@Waechter, constant(3*4+$800))  'Ausgang

DAT
Waechter
  mov pin,par
  and pin,#$ff
  shr pin,2
  mov delta, par
  andn delta, #$ff  wc    ' Carry wenn ein weiteres Bit gesetzt ist
  subx delta, delta         ' alter 8080-Trick :-)

  waitpeq :null, pin
  waitpne :null, pin
:w  lockset semaNummer WC
  if_C jmp :w
  rdlong a, zaehlerAdresse
  add a, delta
  wrlong a, zaehlerAdresse
  lockclr semaNummer

  jmp #Waechter

:null long 0
semaNummer    long 0
zaehlerAdresse long 0
pin res 1
delta res 1
a   res 1
```

Nach oben

Beiträge der letzten Zeit anzeigen: Alle Beiträge | Die ältesten zuerst | Los

neuesthema antwort erstellen

Parallax Propeller Forum Foren-Übersicht -> Assembler-Programmierung Alle Zeiten sind GMT + 1 Stunde

Seite 1 von 1

Gehe zu: Assembler-Programmierung Los

Kleines ASM Tutorial (2)

Autor	Nachricht
deSilva Ass Anmeldungsdatum: 03.06.2007 Beiträge: 375	Verfasst am: 03.07.2007, 21:36 Titel: Kleines ASM Tutorial (2)

Eine komplette Anwendung

Dies hier ist das kleinste (SPIN !) Programm, das etwas direkt auf dem Bildschirm ausgibt, indem der Bildwiederholspeicher für das Objekt TV.spin gesetzt wird.

Code:

```
'' randomPixels  - Beispiel 1
'' Aus deSilvas Propeller Schnipseln (c) 2007
'' v0.14

'' Mess-Schleife über Bildschirmrauschen

' Parameter:
' Hydra-ähnlich (10 Mhz, Video auf Ports 24, 25, 26, 27)
' Normalschirm (20 xTiles = 320 Horizontalpixel, Pixelclock = 8x, non-
interlaced NTSC)
' Teststrobe auf Pin 30
'


CON
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000 + 0000

  x_tiles = 20
  y_tiles = 14

  strobePort = 30

VAR
  word  screen[x_tiles * y_tiles]  ' Verweise auf die jeweils 16
Langworte eines Tiles nach TV.SPIN Konventionen
  long  colors[64] ' Die möglichen Farben
  long padding[16]
  long buffer[x_tiles*y_tiles*16] ' wird auf 16 Worte = 64 byte
synchronisieren (s.u.)

OBJ
  tv    : "tv.spin"

PUB start | i, j, n , dx, dy

  bitMap := @buffer & $ffc0 ' auf 16 Worte Grenze synchronisieren

 'Anfärben des Schirms: Grün, gelb, rot, blau - von oben nach unten
  repeat i from 0 to 63
    colors[i] := $00001010 * (i+4) & $F + $2B060C02

 'Für den Treiber TV die "Tiles" initialisieren
  repeat dx from 0 to tv_hc - 1
    repeat dy from 0 to tv_vc - 1
```

```
      screen[dy * tv_hc + dx] := bitmap >> 6 + dy + dx * tv_vc + ((dy
& $3F) << 10)

'--------------------
' Abschnitt 1: Schirmvorbesetzung
  repeat i from 0 to (x_tiles*y_tiles)*16-1
        long[bitmap][i] := n++
'        long[bitmap][i] := cnt
'        long[bitmap][i] := cnt*cnt
'----------------------

 'start tv
  tv_screen := @screen
  tv_colors := @colors
  tv.start(@tvparams)

'--------------------
' Abschnitt 2: Anwendercode

  dira[0] := dira[strobePort] := 1

  repeat
     repeat i from 0 to constant((x_tiles*y_tiles)*16-1)
         ?long[bitmap][i]

     outa ^= constant(1<<strobePort)        '
'---------------------------------

DAT

bitmap        long 0

tvparams                long    0               'status
                        long    1               'enable
                        long    %011_0000       'pins
                        long    %0000           'mode
tv_screen               long    0               'screen
tv_colors               long    0               'colors
tv_hc                   long    x_tiles         'hc
tv_vc                   long    y_tiles         'vc
                        long    8             'hx
                        long    1               'vx
                        long    0               'ho
                        long    0               'vo
                        long    0               'broadcast
                        long    0               'auralcog
```

Wir wollen auf dem Schirm ein "Rauschen" ausgeben und benutzen dafür den (in SPIN) eingebauten Zufallsgenerator. Der Erfolg ist ernüchternd: Das Muster sieht komisch aus (d.h. der "Zufallsgenerator" ist nur begrenzt ein solcher!) und es geht recht langsam! Wir sehen auf der Hydra die rote LED blinken und die Messung an Pin 30 zeigt ca 7 Hertz.

Warum Pin 30? Nun, Masse und Pins 30 und 31 sind als Stecker rausgeführt und wir können das ohne auf der Hydra zu stochern recht bequem abgreifen.

Wir können im "Abschnitt 1" mit der Vorbesetzung experimentieren und finden, dass nur bei einigermaßen "großen" Werten das Ergebnis zufällig aussieht...

Ich habe das auch mal zum Anlass für einige weitere Messungen genommen (s. im entsprechenden Thread). Das Ergebnis ist dieses Programm:

Code:

```
 Beispiel 2
'...
  repeat

     i:=constant((x_tiles*y_tiles)*16)
     repeat while i>0
         ?long[bitmap][--i]
```

```
      ?long[bitmap][--i]
      ?long[bitmap][--i]
      ?long[bitmap][--i]

    outa ^= constant(1<<strobePort)
```

Wir haben hier ein vorsichtiges "Loop-unrolling" betrieben und daran gedacht, das die "For-Schleife" nur mit einer Konstante als Abbruchsbedingung einigermaßen fix ist. Wenn hier (jedes Mal!) ein Ausdruck berechnet wird, sinkt die Performance ins Bodenlose... Wer's nicht glaubt, kann im ersten "Beispiel 1" ja mal **constant** weglassen... Ansonsten ist übrigens die FOR-Schleife mit oberer Konstante ein wenig schneller als die hier verwendete WHILE-Schleife..

Wir schreiben deshalb ein kleines Assemblerprogramm, dass in einer Schleife den gesamten Bildschirm einmal verändert; Code und Aufruf folgt hier; das Gesamtprogramm findet sich im Anhang.

Das Assemblerprogramm ist so schnell, dass ich es künstlich gebremst habe.

Code:

```
Beispiel 3
...
   dira[0] := dira[strobePort] := 1

  cognew(@loop, @i)

  repeat
     i:=constant((x_tiles*y_tiles)*16)
     repeat while i>0
        waitcnt(cnt+10_000)              ' ein wenig Strom sparen

     outa ^= constant(1<<strobePort) ' toggelt Port 0 und 30 wenn
einmal fertig

DAT
'' LOOP lässt den Schirm einmal "rauschen"
'  Parameter:
'  Global wird BITMAP benutzt (s.u.)als Schirmanfang
'  PAR wird mit einem Pointer auf eine Kommunikationszelle gesetzt,
'     in der 0 steht wenn der COG idlet und sonst die Nummer
'     des letzten veränderten Langworts des Schirms
'  Das Assemblerprogramm läuft von hinten nach vorne und stoppt bei 0

loop org 0
      rdlong loopcount, par wz    '
      if_z jmp #loop              ' Parameter ist Null
:lab  mov addr, loopcount         ' Wir berechnen in ADDR die Adresse im
Bildwiederholspeicher
      sub addr, #1                ' Ärgerlicher Offset
      shl addr,#2                 ' x 4: Wortnummer -> Bytenummer
      add addr, bitmap            ' + Basisadresse

      rdlong pattern, addr        ' das Langwort verändern
      rol pattern, #17            ' ein ad-hoc Zufallsgenerator aus zwei
Befehlen
      xor pattern,cnt
      wrlong pattern, addr

      mov waitreg, #234           ' eine laange Kunstpause..
      add waitreg, cnt            ' ... so bemessen, dass wir den
Schirm mit
      waitcnt waitreg,#0          ' ... 60 Hz verändern... Warum
schneller???

      djnz loopcount, #:lab       ' noch ein Wort?
      wrlong loopcount, par       ' Dem Hauptprogramm melden, dass wir
fertig sind!
      jmp #loop
```

```
pattern       long 0
waitreg       long 0
loopcount     long 0
addr          long 0
bitmap        long 0


tvparams                long    0               'status
....
```

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 05.07.2007, 03:14 Titel:

So kann man einen Stack in Assembler verwalten, und Programme sogar rekursiv aufrufen. Größte Sorgfalt ist aber bei der indizierten Adressierung durch selbst modifizierenden Code angebracht! Man denke immer daran, dass dieses Programm ja MEHRFACH aktiviert wird; niemals also über einen CALL (alias JMPRET) hinweg im Vorgriff "patchen"!

Aus diesem harmlosen Stückchen SPIN...

Code:

```
PUB spinFibo(n)
  if n>2
     return spinFibo(n-1)+ spinFibo(n-2)
  else
     return 1
```

... wird dann dieses "Monster":

Code:

```
DAT
fiboasm
' PAR enthält den Verweis auf zwei Langworte
' [0] Argument für fibo (0: Ergebnis ist berechnet)
' [1] Ergebnis
    mov a, wartezeit
    add a, cnt
    waitcnt a,#0     ' Strom sparen
    rdlong a, par
    tjz a,#fiboasm
  ' organisiere einen Stack
    mov stackP, #stack

    jmpret retaddr, #fibo2  ' resultat = fibo(a)

    ' ergebnis liegt vor
    mov a, par
    add a, #4
    wrlong resultat, a
    mov a,#0
    wrlong a, par
    jmp #fiboasm

fibo2
' if a<3 return 1
    cmps a, #3   wc
    mov resultat, #1
    if_c jmp retaddr


    add stackP, #1    ' zeigt immer auf das letzte belegte Element
    movd :f1, stackP
    add stackP, #1
    movd :f2, stackP
:f1 mov 0-0, retaddr ' push
:f2 mov 0-0, a       ' push Argument

    sub a, #1
    jmpret retaddr, #fibo2 ' call fibo(a-1)
```

```
    movs :f3, stackP
    movd :f4, stackP
:f3 mov a, 0-0    ' hole Argument
                  '.. und ersetze es durch das Resultat
:f4 mov 0-0, resultat

    sub a, #2
    jmpret retaddr, #fibo2 ' call fibo(a-2)

' addiere die beiden Teilergebnisse
    movs :f5, stackP
    sub stackP, #1
:f5 add resultat, 0-0
    movs :f6, stackP    ' fuer den Rücksprung
    sub stackP, #1
:f6 jmp 0-0


wartezeit  long 10_000  ' Wartezeit
retaddr  res 1 'ruecksprungadresse
resultat res 1
a   res 1
stackP res 0

stack res 100      ' bzw. soweit es eben reicht :-)
```

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 22.10.2007, 09:41 Titel:

Hier ein weiteres Stückchen Code - diesmal schon für Fortzgeschrittene. Eine Pulsweitenmodulation auf ALLEN Pins. Kann auch zur Ansteuerung von 30 Servos verwendet werden

So, hier nun Lösung #4!

Es ist ein kleines Kunstwerk geworden; ich habe die Parametrisierung geändert, um flexibler zu sein. Es wird jetzt eine 30 LONG Werte lange Tabelle benutzt, die für jeden der Pins 0..29 eine Pulslänge in Mikrosekunden enthält:
0: unbenutzt
1000: 1 ms
2000: 2 ms
andere Werte (Unter- und Übersteuerung) sind möglich
Die Auflösung beträgt also 10 Bits!
Die Anzahl der gleichzeitig benutzten Ports ist nur durch die Beinchen beschränkt.

Ich bin SEHR zufrieden mit mir

Code:

```
{ 24servos-D: Ein Lernprogramm in 4 Schritten: Der 4. Teil
  ----------------------------------------------------------
  Oktober 2007 by deSilva
  v0.12A : Bug behoben Nov 2007

         Demonstration von Pulsmodulation auf mehreren Kanälen
         (S1) Die Tabelle der Sollwerte wird re-organisiert
         (S2) Es wird auf einen 20 ms Tick gewartet
         (S3) Alle beteiligten Lines werden auf High gesetzt
         (S4) Es wird auf den nächsten eingetragenen  mys- Zeitpunkt
gewartet
         (S5) Der entsprechende Port/die Ports wird/werden
zurückgesetzt
         (S6) (S5) wird wiederholt bis die Tabelle abgearbeitet ist

   Die Leistungsdaten sind wie erwartet beträchtlich; die Auflösung
beträgt etwa 10 bit/ms.
   Die Anzahl der Ports ist nicht beschränkt!
```

```
}

CON
  _CLKMODE = XTAL1 + PLL8X '------ Ändern (nur für DEMO)!!
  _XINFREQ = 12_000_000    '----- Ändern (nur für DEMO)!!!

  _HZ = 96_000_000          '----- Ändern (WICHTIG!!!)

CON
   _anzServos = 30
   _rightVal  = 2000
   _leftVal   = 1000

VAR
  long pulseVariables[_anzServos]  ' Nur für die DEMO benötigt

PUB DEMO | ff

  start(@pulseVariables)

' Testroutine:
' Durchläuft den Bereich 1 bis 9.999 mys im 20 ms Raster
' An Ports 0 und 1 können LEDs OHNE Vorwiderstand angeschlossen
werden,
' die werden dann im Mittel mit < 20mA betrieben

  REPEAT
     repeat ff from 1 to 9999 step 10
       pulseVariables[1] := ff
       pulseVariables[2] := 10000-ff
       WAITCNT(CNT+__20msTicks)
     repeat ff from 9999 to 1 step 10
       pulseVariables[1] := ff
       pulseVariables[2] := 10000-ff
       WAITCNT(CNT+__20msTicks/4)

PUB start(tabelle)
{
   Parameter: Eine 30 = _anzServos) LONGs lange Tabelle mit den
   Pulswerten für 20ms Zyklen. Diese Einträge können danach beliebig
   verändert werden. Der Eintrag N entspricht dem I/O Port N.
   Werte (Mikrosekunde):
       negativ oder 0: Keine Benutzung
       1000:           1 ms
       2000:           2 ms
   Werte außerhalb dieses nominellen Servo-Bereichs sind zulässig
(1....10.000 = 10ms))
}
    COGNEW(@pulseServos, tabelle)


DAT
        org 0
pulseServos

        MOV       ctlTaddr,         PAR

        CALL      #enablePorts

        MOV       aNext20ms,        CNT

:mainLoop
        ADD       Anext20ms,        __20msTicks

        CALL      #sortTab

        WAITCNT   Anext20ms,        #0            ' wait to 20ms

        MOV       DIRA,              rPorts
        MOV       OUTA,              rPorts

'vorbesetzung für "Indexarithmetik"
        MOVD      :modDestTime1,    #sortedTab

        MOVS      :modPorts,        #sortedTab+1
        MOVD      :modDestTime2,    #sortedTab
```

```
:lineLoop
:modDestTime1
        TJZ       0-0,      #:mainloop          ' FERTIG!
:modDestTime2
        WAITCNT   0-0,      #0
'modPorts
        ANDN      OUTA,      0-0      'clear

        ADD       :modDestTime1, ___1024

        ADD       :modPorts,     #2
        ADD       :modDestTime2, ___1024

        JMP       #:lineLoop
' ----- Ende der Hauptschleife ------------------

' berechnet die benutzten Ports (Pulswerte > 0)
enablePorts
        MOV       rPorts,    #0           'clear Active Ports
        MOV       r1,        #_anzServos  'loopcounter
        MOV       r0,        ctlTaddr
        MOV       r3,        #1           'start with Port 0
:loop
        RDLONG    r2,        r0
        CMPS      r2,        #1    WC
        IF_NC OR        rPorts,    r3
        SHL       r3,        #1
        ADD       r0,        #4
        DJNZ      r1,        #:loop
enablePorts_ret
        RET


' Liest durch den 32 LONG ParameterBlock mit deb Oulswerten und
erzeugt die sortierte Tabelle

sortTab
{ Debug Strope for Spcope
        OR    DIRA,          #|<0
        OR    OUTA,          #|<0
}
        NEG   aTimeLimit,    #1
        MOV   r1,            #_anzServos


' AdressVorbesetzung für slest-modifizierenden Code
        MOVD      :modDestA,  #sortedTab
        MOVD      :modDestB,  #sortedTab+1
        MOVD      :modDestC,  #sortedTab


:loopA
        MOV   r2,              #_anzServos
        MOV   aNewPorts,       #0
        MOV   aNewTime,        __10000
        MOV   aServoAddr,      ctlTAddr
        MOV   aPort,           #1

:loopB
''''''        ADD       aServoAddr,  #1   -- Oops, das gehörte hier
nicht hin!!


        RDLONG     aPulseWidth, aServoAddr
        CMPS       aPulseWidth,   #1 WC
        IF_C JMP #:endLoopB

        CMPS       aPulseWidth, aNewTime  WC
        IF_NC JMP #:noNew
        CMPS      aTimeLimit, aPulseWidth  WC
        IF_NC JMP #:noNew
        MOV       aNewTime,    aPulseWidth

        MOV       aNewPorts,   #0
```

```
:noNew
        CMP         aPulseWidth, aNewTime  WZ
        IF_Z OR     aNewPorts,   aPort

:endLoopB
        ADD         aServoAddr,  #4
        SHL         aPort,       #1
        DJNZ        r2,          #:loopB

        TJZ         aNewPorts,   #:quitLoops

        MOV         aTimeLimit, aNewTime

        MOV         r0,         #0
:timesN
        ADD         r0,         __resTicks
        DJNZ        aNewTime , #:timesN

' hier eine echte Multiplikationsroutinehin: r0:= __resTicks*aNewTime


        ADD         r0,         aNext20ms

' store into next sorted Entry

:modDestA
        MOV         0-0,        r0


:modDestB
        MOV         0-0,        aNewPorts

        ADD        :modDestA, __1024
        ADD        :modDestB, __1024
        ADD        :modDestC, __1024

        DJNZ        r1,  #:loopA

:quitLoops

:modDestC
        MOV         0-0,       #0

'beginstrobe Port 0
'        ANDN       OUTA,      #|<0
sortTab_ret
         RET

aPulseWidth  LONG 0
aNewTime     LONG 0
aTimeLimit   LONG 0
aNewPorts    LONG 0
aServoAddr   LONG 0
aPort        LONG 0
aNext20ms    LONG 0


'sortedTab     LONG -1[ _anzServos*2+1]

  ' Constante
__20msTicks      LONG _HZ/1000*20
__1msTicks       LONG _HZ/1000
__resTicks       LONG _HZ/1000_000  ' = Takte für us
__zero           LONG 0
__1024           LONG 1024    ' +2 @ dest position
__10000          LONG 10_000
r0              res 1
r1              res 1
r2              res 1
r3              res 1
r7              res 1
r8              res 1

rWait           res 1
rPorts          res 1
ctlPaddr        res 1
```

```
sortedTab      res _anzServos*2+1
          FIT  496

' Das Ende
```

Edit: Ein Fehler beim Label **loopB** behoben... Es gibt inzwischen etwas verbesserten Code.... PDF mit Erläuterungen folgt gelegentlich...

Zuletzt bearbeitet von deSilva am 12.11.2007, 20:22, insgesamt einmal bearbeitet

Nach oben

robbifan
Fortgeschrittener

Anmeldungsdatum: 24.05.2007
Beiträge: 84

Verfasst am: 12.11.2007, 15:49 Titel:

hallo, du hast hier dieses angegeben :

Code:

```
'' randomPixels  - Beispiel 1
'' Aus deSilvas Propeller Schnipseln (c) 2007
'' v0.14

'' Mess-Schleife über Bildschirmrauschen

' Parameter:
' Hydra-ähnlich (10 Mhz, Video auf Ports 24, 25, 26, 27)
' Normalschirm (20 xTiles = 320 Horizontalpixel, Pixelclock = 8x, non-
interlaced NTSC)
' Teststrobe auf Pin 30
'

CON
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000 + 0000
```

_xinfreq = 10_000_000 + 0000 , heisst das, man muss einen 10mhz-quarz haben?

wie kann ich es mit einem 5mhz laufen lassen?

warum wird das mit 10mhz hier im forum gemacht?
der standard ist doch 5mhz beim propeller , oder?

mfg

Nach oben

ErNa
Fortgeschrittener

Anmeldungsdatum: 28.06.2007
Beiträge: 149

Verfasst am: 12.11.2007, 16:34 Titel:

Ja, 5MHz sind der Standard. Wenn jemand einen anderen Quarz hat und verwenden will, dann muss er natürlich das dem Programm mitteilen.

Bitte nicht ärgerlich sein- wenn jemand Codeschnitzel bereitstellt, ist das grundsätzlich positiv.

Nach oben

Ariba
Fortgeschrittener

Anmeldungsdatum: 07.04.2007

Verfasst am: 12.11.2007, 18:31 Titel:

robbifan hat Folgendes geschrieben:

Beiträge: 80

wie kann ich es mit einem 5mhz laufen lassen?

warum wird das mit 10mhz hier im forum gemacht?
der standard ist doch 5mhz beim propeller , oder?

mfg

Hydra und SpinStamp verwenden einen 10MHz Quarz, darum ist das ein zweiter Standard.

Um es mit einem 5MHz Quarz laufen zu lassen, musst du nur die Constanten anpassen:

Code:

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
```

dies setzt PLL*16:
5*16=80MHz genauso wie 10*8=80MHz zuvor

Gruss
Andy

Nach oben

deSilva
Ass

Verfasst am: 12.11.2007, 20:25 Titel:

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Auf den meisten Boards ist der Quarz steckbar, sogar auf der Hydra und dem Protoboard!

Ich benutze inzwischen gerne 6 MHz, einfach so 

Das größere Kompatibiltätsproblem sind die verschiedenen Ports und die völlig andere Hydra PS/2 Schnittstelle..

Nach oben

robbifan
Fortgeschrittener

Verfasst am: 13.11.2007, 13:01 Titel:

Anmeldungsdatum: 24.05.2007
Beiträge: 84

...._clkmode = xtal1 + pll16x....

was ist das denn? warum vedoppelt sich dieser wert wenn die taktfrequenz auf 5mhz läuft?

Nach oben

ErNa
Fortgeschrittener

Verfasst am: 13.11.2007, 13:28 Titel:

Anmeldungsdatum: 28.06.2007
Beiträge: 149

Nein, bei 5 MHz 16, bei 10 MHz 8.
Damit kommt man auf 80 MHz internen Takt.
Hat man einen 6 MHz Quarz, so stellt man 16 ein, der Prozessor ist dann allerdings etwas übertaktet!

Nach oben

deSilva
Ass

Verfasst am: 14.11.2007, 02:41 Titel:

Anmeldungsdatum: 03.06.2007
Beiträge: 375

@Robbi: Das steht alles im Kapitel 1 des Manuals...

Nach oben

Beiträge der letzten Zeit anzeigen: Alle Beiträge | Die ältesten zuerst | Los

neuesthema antwort erstellen

Parallax Propeller Forum Foren-Übersicht Alle Zeiten sind GMT + 1 Stunde
-> Assembler-Programmierung

Seite 1 von 1

Gehe zu: Assembler-Programmierung | Los

Du **kannst keine** Beiträge in dieses Forum schreiben.
Du **kannst** auf Beiträge in diesem Forum **nicht** antworten.
Du **kannst** deine Beiträge in diesem Forum **nicht** bearbeiten.
Du **kannst** deine Beiträge in diesem Forum **nicht** löschen.
Du **kannst** an Umfragen in diesem Forum **nicht** mitmachen.
Du **kannst** Dateien in diesem Forum **nicht** posten
Du **kannst** Dateien in diesem Forum **nicht** herunterladen

Timer und Counter - ASM-Tutorial (3)

neues Thema antwort erstellen Parallax Propeller Forum Foren-Übersicht -> Assembler-Programmierung

Vorheriges Thema anzeigen :: Nächstes Thema anzeigen

Autor	Nachricht
deSilva Ass Anmeldungsdatum: 03.06.2007 Beiträge: 375	Verfasst am: 05.08.2007, 14:17 Titel: Timer und Counter - ASM-Tutorial (3)

Timer und Counter

Der Propeller hat 16 davon und zwar 32 Bit breite. Ich wüsste kein anderes Teil mit dieser Ausstattung! Aber wie auch bei den MIPSen (8x20 = 160) ist dies ein eher theoretischer Wert: Die meisten COGs warten einfach nur, und was die Timer angeht, so sind COGs meist auch mit anderen Aufgaben betraut. Aber wenn es „hart auf hart" gehen sollte, dann stehen eben 16 Timer/Counter zur Verfügung.

Wenn ich immer Timer/Counter schreibe, dann ist hier schon eine Doppelfunktion erkennbar. Diese Hardware kann einerseits externe an einen Pin angelegte Signale „zählen", andererseits auch den interne Takt (**TIMER**). Wozu ist letzteres nützlich? Denn den internen Takt können wir auch im CNT Register ablesen! Nun, typischerweise gibt es 3 Anwendungen:
- Man kann in Abhängigkeiten von Überläufen dieser „Uhr" einen Interrupt auslösen („Wecker"); das ist so beim Propeller allerdings nicht möglich.
- Man kann für eine andere Hardware-Unit Signale zur Verfügung stellen (UART-Takt, A/D-Wandlerzeit,...); beim Propeller wird diese Betriebsart für die Video-Logik genutzt
- Man kann in Anhängigkeit von dieser Uhr ein Signal an einem Ausgangspin erzeugen. Beim Propeller kann das sein
o Rechteckimpulse bestimmter Frequenz
o PWM (Oneshot)
o DUTY Signal

Zusätzlich kann der Propeller gleichzeitig das invertierte Signal an einem anderen Ausgangs-Pin ausgeben; das spart möglicherweise einen Inverter).

In der Betriebsart **COUNTER** hingegen werden Zustände an einem oder zwei Eingangspins „gezählt". Eine gute Frage wäre jetzt, was man eigentlich unter „zählen" versteht. Wird da ein (schmaler) Puls gezählt? Auch. Aber in Wirklich wird – wie im Timer Modus – gezählt, wie viele Prozessortakte ein bestimmter Zustand an den Pins herrscht. (Im Timer Modus wird einfach nur „immer" gezählt).

Wenn es sich also um eine "Flanke" handelt, bei der von einem Takt zum nächsten der Übergang erfolgt (der erfolgt IMMER in einem Takt) dann zählt das wie erwartet "Eins".

Die möglichen Zustände, IN DENEN gezählt wird (NICHT: die gezählt werden!) sind:

- A ist high
- A ist low
- A ist high und war einen Takt vorher low (positive Flanke)
- A ist low und war einen Takt vorher high (negative Flanke)

- Eine beliebige logische Kombination von high/low Zuständen zweier Pins A und B (hiervon gibt es 10 interessante; formell sind es 16)

Vielleicht hat sich eben schon jemand gefragt: Duty-Cycle, PWM,... hmmm, ist das nicht irgendwie dasselbe???

Das habe ich mich in der Tat auch gefragt Das dahinter steckende Problem ist: Der Propeller kann „von alleine" gar kein PWM-Signal erzeugen, sondern nur mit einer zusätzlichen Schleife durch den Programmierer! Der sogenannte PMW-Modus ist (nur) die hierfür notwendige Unterstützung. Genau gesagt wird am Anfang einer Periode der Länge P = 2**32 / CLKFREQ / FRQA Sekunden (oder auch kürzer) ein Puls der Länge L = -PHSA / CLKFREQ / FRQA ausgegeben – es handelt sich also eher um ein „Monoflop" (Oneshot). Man muss nun zusätzlich dafür sorgen, dass jedes Mal nach Ablauf von P das Register PHSA wieder auf den richtigen (negativen!) Wert gesetzt wird.

Was passiert technisch? Nichts weiter, als dass das Bit 31 (="Vorzeichen") von PHSA ausgegebenen wird!

Anders der DUTY-Mode; der läuft vollständig alleine: Allerdings wird hier ein schmaler Puls (12,5 ns) mit unterschiedlicher Frequenz CLKFREQ*FRQA/2**32 ausgegeben. Der Duty-Cycle stimmt da zwar, aber so ein „Spike" kann schon mal in einem Low-Pass verloren gehen. Und ein Duty Cyle von ½ wird brutal mit 40 MHz realisiert...

So jetzt fehlen eigentlich nur noch ein paar Beispiele - in Assembler natürlich ... Kommen später, wenn es wieder Regenwetter gibt :-)

Nach oben

Beiträge der letzten Zeit anzeigen: Alle Beiträge | Die ältesten zuerst | Los

neues thema | antwort erstellen

Parallax Propeller Forum Foren-Übersicht -> Assembler-Programmierung

Alle Zeiten sind GMT + 1 Stunde

Seite 1 von 1

Gehe zu: Assembler-Programmierung | Los

Propeller Video ohne Geheimnisse

neues thema antwort erstellen Parallax Propeller Forum Foren-Übersicht -> Allgemein

Vorheriges Thema anzeigen :: Nächstes Thema anzeigen

Autor	Nachricht
deSilva Ass Anmeldungsdatum: 03.06.2007 Beiträge: 375	Verfasst am: 22.07.2007, 09:48 Titel: Propeller Video ohne Geheimnisse

(1) Grundlagen

Als Gerät ist ein Fernseher wohl das dümmste, was man sich vorstellen kann, schließlich ist die (jedenfalls „Schwarz-Weiß") Fernsehtechnologie an sich um 1928 festgeschrieben worden (damals mit 30 Zeilen bei 12,5 Herz Bildwechsel!), und die Gedanken dazu („Nipkow") sind nicht lange nach der Erfindung des Morsecodes entstanden... Dümmer als Fernsehgeräte sind eigentlich nur noch VGA-Monitore.

Denn man muss ihnen ALLES sagen, nicht einmal, nicht hundertmal - nein, 25 mal in jeder Sekunde (den amerikanischen sogar 30 mal). „Frohf" wird eventuell hierzu weiteres Interessante schreiben, wir gucken uns jetzt nur mal schnell das von „Ariba" gemachte Oszillogramm http://propellerforum.sps-welt.de/viewtopic.php?t=638&postdays=0&postorder=asc&start=25 an. (Wir werden da erst im 3. Teil dieser Serie weiter drauf eingehen.) So ein Signal muss also erzeugt werden: Jedes „Pixel" jeder Zeile wird durch einen Helligkeits- („Luma") und ggf. Farbwert („Chroma") definiert.

Was heißt denn überhaupt „Pixel"? In der Frühzeit der Fernsehtechnik lief ein Elektronenstrahl von links oben nach rechts unten über das mehr oder weniger flache Ende eine Phosphor beschichteten Vakuumröhre, genauso, wie man eine Buchseite schreibt oder liest. Die Helligkeitsstufen ergab sich durch die Stärke dieses Strahls und die „Auflösung" durch die Grenzen, seine Größe zu fokussieren.

Das „Videosignal" wurde direkt an das (ein) Gitter dieser Röhre geführt, und so versteht man vielleicht am Besten, warum es so aussieht. Das ganze musste natürlich noch „synchronisiert" werden, aber das ist eine andere Geschichte, die „Frohf" irgendwann mal erzählen wird

Mit der Erfindung des Farbfernsehens und später der TFT Schirme wurde aber eine Diskretisierung zwingend: Die Auflösung wurde durch die Anzahl der auf der Röhre angebrachten Farbpunkte, bzw die Transsistoren auf dem TFT Schirm bestimmt. Auf Grund der Geschichte des Schwarz-Weiss Fernsehens gab es hier Randbedingungen: Ein Farbfensehbild durfte – für den Preis! – natürlich nicht „schlechter" sein als eine schwarz-weißes. Die Anzahl dieser Punkte beträgt in Amerika (und einem nicht unerheblichen Teil der übrigen Welt (*NTSC M-Standard*))

720 x 486

Dies soll uns nur zur Orientierung dienen, „Europäische" Standards sehen anders aus und das 16:9 Format und HDTV sowieso; aber nahezu jedes kleine (gerade die!) und größere Fernsehgerät „versteht" NTSC, u.a. weil dies der Standard in Japan und Korea

ist...

Wir müssen jetzt noch auf eine technische Finesse eingehen, die „Interlacing" genannt wird. „30 mal" (ganz genau: 29,97 mal) pro Sekunde wird ein vollständiger Bildschirminhalt also gesendet: Flimmert das nicht? Aber wie! Natürlich könnte man die „Beschichtung" so wählen, dass das Bild „nachleuchtet" – das hat jeder sicher schon einmal an Radarschirmen gesehen. Aber für bewegte Bilder ist dies keine wirkliche Alternative. Die gefundene – in der Tat raffinierte! – Abhilfe besteht darin, das Fernsehbild in zwei Gruppen ("Halbbilder") – die 243 ungeraden und die 243 geraden Zeilen - zu teilen, und diese nacheinander – also dann im 1/60 Sekunden Abstand - zu schicken. Dadurch kann man „ein bisschen" das Nachleuchten der Zeilen des letzten Halbbildes ausnutzen und das Flimmern ist für die Meisten nur noch aus den Augenwinkeln bemerkbar. Diese Trick heißt also „Interlacing", in gutem technischen Deutsch „Zeilensprungverfahren".

243 × 320

Das könnte uns eigentlich ziemlich wurscht sein, wenn sich nicht die Hersteller von Billiggeräten (und Kleinstmonitoren) gedacht hätten: Das ist doch nur redundant! Und beherzt nur noch 243 Transistorzeilen auf ihre TFT Schirme löten (aus unerfindlichen Gründen wird allerdings fast immer 234 angegeben... Vielleicht sollte da wirklich mal jemand nachzählen). Da nun die „Pixel" nicht mehr quadratisch wären, macht man dann gleich Nägel mit Köpfen und halbiert auch die Anzahl horizontaler Bildpunkte von 720 auf weniger als die Hälfte (= 320).

Dies sind die Geräte, die man für unter 100 Euro bekommt (allerdings immer häufiger etwas breitere Schirme (16:9), so dass 480 quadratische Pixel in die Zeile passen.)

Dies ist die Situation, mit der wir zurechtkommen müssen, wenn wir qualitativ erträgliche Videoausgabe erzeugen wollen! Es hat überhaupt keinen Zweck (und führt i.d.R. sogar zur Verschlechterung!), wenn wir mehr Pixel in das Videosignal hineinbringen, als das angeschlossenen Gerät rafft! Das ist bitter, weil man ja häufig nicht nur einen Monitor anschließen will, und natürlich die „Gestaltung" des Bildschirmaufbaus auch von der Größe anhängt.

- Wollen wir auf eine Zeilenlänge 480 gehen, auch auf die Gefahr hin, dass ein Monitor das dann nicht mehr fängt ? (Wir können aber das Framing leicht umkonfigurieren, dazu später mehr)
- Wollen wir eine doppelte (= eigentlich "normale") Zeilenauflösung (="interlaced") benutzen? Die kleinen Geräte haben diese Zeilen einfach nicht, und mischen die Halbbilder mit fragwürdigem Ergebnis zusammen..

In Arbeit
(4) Farbe und die Grenzen der Propeller Hardware
(5) Ein ganz einfacher Video-Treiber in Assembler(Hydra HEL-VIDEO)
(6) „Tiles" und „Sprites" vs. voller Bitmap
(7) VGA oder TV ?

Zuletzt bearbeitet von deSilva am 01.08.2007, 22:48, insgesamt 2-mal bearbeitet

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 22.07.2007, 21:59 Titel:

(2) Textausgabe mit einem modifizierten TV_TEXT

Bevor wir uns ausführlicher damit beschäftigen „wie genau" der Propeller so ein benötigtes „Videosignal" erzeugt, wollen wir uns die Benutzung eines Treibers – in diesem Fall TV.SPIN an einem Beispiel angucken. Im einfachsten Fall soll der Propeller eine Alfanumerische Anzeige machen, wie man es von den üblichen Millionen 1x16 bis

4x40 Billiganzeigen gewöhnt ist. Hierzu programmieren wir erst mal eine Uhr, da wir nicht so einfach an die wirkliche Uhrzeit rankommen (DCF-77 Modul ca 12 Euro, GPS „Maus": 60 Euro) programmieren wir eine Stoppuhr (Übungsaufgabe für den Leser: P0 eine LED, P1 eine Taste: 1. Druck START, 2. Druck LAP-Zeit, 3. Druck: auf 0 setzen; zwischen 1. und 3. Drücken soll die LED zu Kontrollzwecken leuchten...)

Hier nur der Basiscode dafür:

Code:

```
CON
  '
  'Hydra
  _clkmode = xtal1 + pll8x          ' enable crystal and pll times 8
  _xinfreq = 10_000_000 ' set frequency to 10 MHZ
  _VPIN   = 24
  '
  'MPE-V1
  _clkmode = xinput + pll16x          ' enable external clock
  _xinfreq = 5_000_000
  _VPIN = 16                          ' erster der 3 Videoausgänge
OBJ
  DISP : "MPE_TEXT_010.SPIN"

PUB stoppuhr | refZeit, dieZeit, dieStunde, dieMinute, dieSekunde,
dasZehntel, _zehntelsek

' Hier die Steuerzeichen für TEXT
'     $00 = clear screen
'     $01 = home
'     $02 = clear screen with color (mod MPE)
'     $08 = backspace
'     $09 = tab (8 spaces per)
'     $0A = set X position (X follows)
'     $0B = set Y position (Y follows)
'     $0C = set color (color follows)
'     $0D = return

' ... und hier die Farbpalette
'0    white / black
'1    black / white
'2    gray  / black
'3    black / gray
'4    gray / white
'5    white / gray
'6    white / black


  _zehntelsek := clkfreq/10

  DISP.start(_vPin, 16)  ' 16 Zeichen pro Zeile, damit es schön groß
wird
  DISP.str(string(11,6,10,1,12,6,"Stoppuhr v0.2 "))

  DISP.str(string(10,3,11,2,12,3,"HH:MM:SS,Z"))  ' Pos: 4. Spalte, 3.
Zeile

  dieZeit := cnt

  repeat
    dieZeit += _zehntelsek
    waitcnt(dieZeit)
    if ++dasZehntel => 10
        dasZehntel -= 10
        if ++dieSekunde => 60
           dieSekunde -=60
           if ++dieMinute => 60
              dieMinute -=60
              ++dieStunde

    DISP.str(string(10,3,11,3, 12,1))  ' Pos: 4. Spalte, 4. Zeile
    DISP.dec(dieStunde,2)
    DISP.out(":")
    DISP.dec(dieMinute,2)
```

```
    DISP.out(":")
    DISP.dec(dieSekunde,2)
    DISP.out(",")
    DISP.dec(dasZehntel,1)
```

Um den eigentlichen Grafiktreiber TV.SPIN wollen wir uns erst mal nicht kümmern, sondern um TV_TEXT, das ich für unsere Übungszwecke etwas umgeschrieben habe – in der Anlage MPE_TEXT_010. Dieser „Text Treiber" besteht aus drei Teilen:

(1) Einer Initialisierung des TV.SPIN Treibers
(2) Der Verwaltung einer „Tile Map" mit dem Namen SCREEN
(3) Einigen „Utilities" zur Ausgabe von ganzen Strings, Dezimal- und Hexadezimalzahlen, sowie zum Setzen der Vordergrund-/ Hintergrundfarbe eines Zeichens

Die Initialisierung von TV.SPIN erfolgt über einen Parameterblock TV_PARAMS aus 14 Worten; die Detailinformation über die genaue Länge der einzelnen Signalkomponenten ist hier jedoch **nicht** definiert, sondern wird nur durch ein Bit PAL/NTSC (ggf noch „Interlaced") angegeben. Aber der Aufbau der „Nutzinformation" kann beeinflusst werden. Wir erinnern uns: Die Nutzinformation (NTSC-M) beträgt bestenfalls 720 Spalten und 243 non-interlaced Zeilen. Der TV-Treiber rechnet grundsätzlich in Einheiten von 16, sowohl „hoch" wie „weit"; Letzteres hängt wesentlich mit der später behandelten Hardware zusammen, Ersteres ist einfach so. Wir nennen so ein Quadrat eine Kachel (=„Tile"), und überschlagen kurz, dass wir da maximal 15 x 45 Kacheln unterbringen können. Diese beiden Zahlen sind in den Parametern TV_COLS (alias HC) und TV_ROWS (alias VC) einzutragen.

Wir müssen aber nicht so viele – ach, es sind ja so wenige! – Kacheln unterbringen; weniger Kacheln werden vom TV-Treiber zentriert - oder auf Wunsch auch gezielt links bzw. oben – angeordnet (Parameter: HO und VO: horizontal bzw vertical offset). Wenn wir weniger Kacheln benutzen, können wir sie möglicherweise auch „vergrößern", dass geht allerdings jeweils nur um den Faktor zwei. Dieses Verfahren bietet sich hier an, weil unsere 45 x 15 Kacheln ja keineswegs „Bit-Quadratisch" sind, und wir ja auch meistens gar kein Ausgabegerät mit 720 echten Pixeln pro Zeile haben. Wir „verdoppeln" also sinnvoller weise in der Horizontalen! Der MPE_TEXT-Treiber macht das in der START Routine automatisch unter Berücksichtigung des 2. Parameters COLREQUEST (den man dann sinnvoller weise auf 20 bis 22 setzen sollte.) Das gibt dann das optimale Bild für die kleinen Monitore: Nicht viel drauf - das aber deutlich!

(Übungsaufgabe: TV mal so parametrisieren, dass nur EINE EINZIGE Kachel angezeigt wird!)

Wir würden nun erwarten, dass diese Kacheln in einer „üblichen Anordnung" in einem Bildwiederholspeicher vorliegen, so wie auf kariertem Papier die Karos angeordnet sind: links oben das erste und rechts unten das letzte.
Weit gefehlt! Der TV-Treiber sucht sich diese Kacheln von (fast) BELIEBIGEN Adressen aus dem Speicher zusammen, nicht nur aus dem RAM sondern auch aus dem ROM. Das ist nämlich der Clou, wenn wir (nur) Zeichen ausgeben!

Anstatt jedes Mal die einzelnen Pixel für die Zeichen an die richtige Stelle im Bildwiederholspeicher zu setzen, verwenden wir nur eine Adresse (oder auch eine „Codenummer") , wo diese Pixel zu finden sind, und bei der Signalerzeugung werden sie dann zusammengesucht. Dies Technik wurde früher beim Bau von ASCII-Terminals verwendet, ursprünglich auch ohne jedwede Software dabei. Für die Verwaltung von 20 x 15 Zeichen brauchen wir also zwar 300 Kacheln, aber wir können mit nur 300 Pointern darauf verweisen. Diese Tabelle wird in den meisten Programmen – vielleicht etwas irreführend – SCREEN genannt. Der TV-Treiber erwartet diese Adressen in 16-Bit-Worten (was ja eigentlich für 64 KB ausreicht; da die Kacheln aber sowieso eine Größe von 16 LONG-Worten = 64 Bytes haben, wird hier nur eine „Kachelnummer" verwendet, das ist die um 6 Bit verschobene Adresse. Also Achtung: Kacheln müssen immer auf 64 Byte

die um 6 Bit verschobene Adresse. Also Achtung: Kacheln müssen immer auf 64 Byte Grenzen liegen. Das ist total lästig für selbstdefinierte Zeichen, weil man sich so etwas nicht ohne weiteres vom Compiler wünschen kann....

Die oberen 6 Bits werden dann für etwas Anderes verwendet – das „kriegen wir später".

Doch für den Normalfall hat Parallax 256 wunderbare hochaufgelöste (nicht: hochauflösende J) Zeichen in den ROM gepackt, beginnend auf Adresse $8000 (= „Kachelnummer" $200): Da steht „ASCII- Null".

Diese wunderbare Tabelle hat nur zwei dumme Eigenschaften: Die Zeichen sind nicht 16 Zeilen hoch sondern 32; und sie sind auch nicht 16 Pixel breit... Bzw. das sind sie schon, aber man sieht es nicht auf den ersten Blick.

Aufmerksame Leser haben ja schon vor 5 Minuten ein Posting geschrieben: „DeSilva, Du hast da einen Rechenfehler gemacht: Eine Kachel aus 16x16 Pixeln benötigt ja nur 32 Bytes und nicht 64!"
Wohl war, aber wer hat denn gesagt, dass 1 Pixel = 1 Bit sei? Für ein Pixel verwenden wir nämlich ZWEI Bits, um diesem Pixel eine „Färbung" zu geben, d.i. einen Grauwert oder eine „richtige" Farbe: Hierfür haben wir also offenbar 4 Möglichkeiten.....

Fassen wir noch mal zusammen, bevor die Verwirrung galaktische Ausmaße annimmt:
(1) Auf dem Bildschirm wird vom TV-Treiber eine Menge von 16x16 Pixel großen Kacheln dargestellt, die einzelnen 256 Pixel jeder Kachel– wir haben die Katze ja schon aus dem Sack gelassen – können eine von vier verschiedenen Färbungen haben. Deshalb besteht eine Kachel im Speicher aus 32 Bit x 16, jeweils zwei aufeinanderfolgende Bits in einem Wort bestimmen ein Pixel.
(2) Wenn wir Zeichen aus der Character-Map im ROM verwenden wollen, dann wird dort nur jedes zweite Bit in den (LONG) Worten zur Zeichendefinition verwendet. Jedes andere zweite ist beliebig (konkret: beschreibt das folgende/vorhergehende Zeichen).
Bei der Ausgabe, d.i. bei der „Wandlung" unserer Bit-Kacheln in Signalpixel, können wir nun folgendes tun:
a. Jedem der Pixel eine (aus einer Auswahl von 4, s.o.) andere Färbung geben (Übungsaufgabe!)
b. Nur zwei Farben verwenden (Zeichenfarbe, Hintergrundfarbe); Beispiel:

Code:

```
Zwei-Bit-Wert   Farbe
    0:          BLAU
    1:          WEISS
    2:          BLAU
    3:          WEISS
```

Hierdurch „ignorieren" wir - quasi automatisch - ein Bit, und das richtige Zeichen erscheint weiß auf blau. Dies ist der Trick, der den Erfindern der Character Map im Sinn stand, und MPE_TEXT (und das Original TV_TEXT) nutzen das so auch aus.

So, da war doch noch was? Richtig: Die Zeichen sind zu allem Überfluss auch noch 32 Zeilen hoch, passen also vertikal nicht in eine „Kachel". Nun ist das in diesem Fall nur eine Treiberkonvention (im Gegensatz zur 4-Farb-Angelegenheit, die Hardwaregründe hat.)

Da ich für dieses Kapitel nicht den TV-Treiber ändern wollte, habe ich zu einem ganz billigen Trick gegriffen: Ich verwende jeweils zwei Kachel-Zeilen für jede **logische** Zeile. Beim Lesen des Codes sticht das ins Auge durch verschiedene Multiplikationen mit zwei. Das heißt aber auch: Wir haben nicht mehr 15 Zeilen, sondern nur noch 7, um die Zeichen ihrer vollen Schönheit auszugeben.

Der Original TV_TEXT Treiber umgeht dieses Problem, da er nur im „Interlaced Mode" arbeitet, d.h. akzeptabel nur auf einem Monitor oder Fernseher mit (mehr als) 480 „echten" Zeilen.

==========

Wir müssen jetzt noch eine Angelegenheiten besprechen, die das gesamte Propeller Video Konzept – je nach Ansicht - zu einem „Fass ohne Boden", zu einer „Intellektuellen Herausforderung", zu einer „Vergeudung eines riesigen Intelligenzpotenzials für die Umgehung selbstverursachter Schwierigkeiten" oder zu einer „Katastrophe schlechthin" macht: Farben und Helligkeitsstufen.

Wir sind es heute gewöhnt, zwischen 64 Tausend und 8 Millionen Farben wählen zu können, der Eine oder Andere erinnert sich vielleicht auch noch an 256 Farbpaletten. Hardware bedingt, können am Propeller bei der Benutzung eines COGs im VGA-Modus nur 64 Farben verwendet werden, damit könnte man leben, wenn nicht die ganze Welt Composite Video verwenden würde, und hier ist die Lage bedeutend komplexer. Man muss den Parallax Entwicklern zugestehen, dass sie mit wenig Aufwand viel erreicht haben, aber auf Kosten der Anwender, die immer wieder Umgehungen der viel zu engen Einschränkungen suchen müssen.
Was genau sind nun diese Einschränkungen?
Nun, für jede 16 Pixelfolge können diese 16 Pixel nur jeweils eine von 4 vorgegebenen Farb- (bzw. Helligkeits-)werten annehmen. (BTW: Wir haben die zusätzliche Einschränkung kennen gelernt, dass sogar für jede Kachel aus 16x16 Pixeln die Pixelfärbung nur aus dieser Menge gewählt werden kann; das ist jedoch nur eine Software Konvention des beliebtesten Treibers TV.SPIN, die man leicht überbrücken kann.)
Allerdings kann jede Kachel andere 4 Farben zu Grunde legen aus einem bedeutend größeren Farb- und Helligkeitsraum. Dies ist genau die Aufgabe der 6 oberen Bits in den SCREEN-Wörtern, deren unteren 10 Bits die entsprechende Kachelnummer angeben: Ein Verweis auf eines von 64 LONG-Wörtern (ich nenne sie jetzt mal: FARBWORTE), in denen die möglichen 4 Farben dieser Kachel kodiert sind! (Der Vektor mt diesen Farbworten heißt meist COLOR.)

Dies ist natürlich auch nur eine Software-Konvention... Wenn man mehr als 64 „Farbzusammenstellungen" braucht, dann kann man sich das natürlich umbauen.

Allerdings kann man nichts daran ändern, welche **Grundfarben** überhaupt zur Verfügung stehen (die in jeweils 8 Bit der angesprochenen Farbworte kodiert sind.)

Für unser erstes Beispiel MPE_TEXT_010 habe ich absichtlich auf Farben verzichtet, und nur Helligkeitswerte verwendet: weiß, grau, schwarz. Die Palette besteht nur aus 6 (!) verschiedenen Zusammenstellungen (weiß auf schwarz, schwarz auf weiß, grau auf weiß, weiß auf grau, grau auf schwarz, schwarz auf grau). Wir erinnern uns: Wegen der trickreichen Überlappung von zwei Zeichen kann nicht jedem einzelnen Pixel eine Farbe gegeben werden.

Diese Palette kann man aber beliebig ändern und auch echte Farben hinzufügen; hierzu ist folgende Tabelle hilfreich:

Code:

```
Farbbyte (8 Bit) : 4 Bit „Chroma"  |  1 Bit „Chroma Enable"  | 3 Bit
„Luma"

Luma:
   0: Schwarz
   1: d'grau
   2: h'grau
   3: fast weiß
   4: ziemlich weiß
Auf 0 und 7 sollte man möglichst verzichten (Erklärung hierfür
```

```
Chroma Enable sollte 0 sein für reine Grauwerte

Colors:
      0: Blau
      5: Rot
...   15:
```

Wir können also 16 Farbnuancen + Schwarz/Weiß in ca 5 Helligkeitsstufen darstellen = 85 „Farben" Das ist so schlecht nicht.

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 29.07.2007, 22:00 Titel:

Wieder was gelernt! Bei Experimenten wird schon mal gelegentlich das "Chroma-Bit" in der tv_mode Konfiguration gesetzt. Auch ich hatte das getan, weil ich ja einen Schwarz-Weiß Treiber machen wollte. Zur Sicherheit eben...

Doch der Effekt ist verblüffend. Das gesetzte Bit bewirkt ja in erster Linie die "Isolation" des Farbsignals auf den 4. Pin (für S-Video)

Doch auf manchen Boards - z.B. der Hydra - ist dieser Pin mit einem 470 Ohm Widerstand an die Video-DAC angekoppelt. Das dient an sich dem Einsteuern des Tonsignals in der Breitbandvariante.

Doch nun wandert ein trägerloses Farbsignal in das Videosignal und führt zu einem deutlich wahrnehmbaren "Farbrauschen"!

Also bitte das entsprechende Bit in TV_MODE wegmachen! Ich hatte das nicht gemerkt, weil meine Bastelanlage ja keine vier Widerstände hat - zeitweise bekanntlich ja nicht mal drei 😉

Zuletzt bearbeitet von deSilva am 02.08.2007, 09:08, insgesamt einmal bearbeitet

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 01.08.2007, 22:15 Titel:

(3) Die Video-Logik (Allgemein)

Wir wollen uns der Video-Logik im Propeller ganz vorsichtig nähern, weil sie nämlich auch für andere Dinge nützlich sein kann
- allgemeine 8 Bit Ausgabe
- 8 Bit DAC mit R2R Netzwerk

Die Videologik hat drei Operationsmodi
- Composite Video (Baseband)
- Broadband Video
- VGA

Wir gucken uns zuerst den VGA-Modus an (Dieser Modus wird im **VCFG**-Register ("Video Configuration") eingestellt (VMode = 01)); Composite Video ist „ähnlich".

Für alle Modi gibt es ein steuerndes Schieberegister (PIXELS) und ein Datenregister (COLORS); diese Namen sind nicht offiziell, sondern gerade von mir erfunden worden :-) Wir können uns vorstellen, dass mit einer gleich zu besprechenden Clock die Bits aus dem PIXELS-Register rausgeschoben werden – und zwar das LSB zuerst - entweder eines oder zwei zur Zeit; diese beiden Möglichkeiten heißen 2-Farb bzw. 4-Farb-Mode; wir werden nachher verstehen, warum.

O.k. was geschieht nun mit den 32 jeweils heraus geshifteten Bits oder den 16 Bitpaaren? Wohin werden sie geshiftet?

Eine gute Frage! Sie erscheinen nicht wie erwartet an einem I/O-Pin sondern werden zur Adressierung eines der vier Bytes (!) im COLORS Register verwendet. Die 32 bit dieses Registers werden nämlich in 8 Bit Gruppen aufgeteilt, von rechts (LSB) nach links (MSB) nennen wir diese Bytes mal A, B, C und D. Die Zuordnung ist nun - wie erwartet- :

Code:

```
2-Bit-Wert      COLORS
0               A
1               B
2               C (nur im 4-Farben-Modus)
3               D (nur im 4-Farben Modus)
```

Genau diese 8 Bits A, B, C oder D werden an konfigurierbaren 8 I/O Pins ausgegeben (also an 0..7, 8..15 oder 16..23). Und das ist eigentlich schon (fast) alles!

Die Werte für PIXELS und COLORS werden im WAITVID-Befehl gewählt (zwei Parameter).

Will man ein bestimmtes 8-Bitmuster ausgeben, dann muss man demnach folgendermaßen vorgehen:

Code:

```
WAITVID (achtbitmuster, 0)
```

Jetzt wird 32 mal das „achtbitmuster" - die rechten 8 bits davon -ausgegeben. Das ist natürlich nicht schrecklich spannend; wir wollen wahrscheinlich eher bei jeder Schiebeoperation ein anderes Muster ausgegeben, z.B. $FF, $1F, $07, $00

Nahe liegend ist diese Lösung (Im 4-Farb-Mode natürlich!):

Code:

```
WAITVID ( $FF1F0700, %%0123 )
```

Das Problem ist nun, dass danach 12 Clocks lang 0 ausgegeben wird; wir können natürlich leicht 4 „Sägezähne" erzeugen:

Code:

```
WAITVID ( $FF1F0700, %%0123012301230123 )
```

Doch es gibt eine weitere Einstellung („FrameClock"), die angibt, wie viele der 32 PIXELS überhaupt verwendet werden sollen: Im zweiten Videosteuerregister **VSCL** (=„scale") setzen wir den Wert FrameClock z.B. auf 4 (im 4-Color Mode werden dann (die rechtesten) 8 Bit verwendet).

Was ist nun der Vorteil gegenüber dem Setzen mit OUTA? Eigentlich nur, dass es im Prinzip etwas schneller geht, und dass man sich nicht besonders um das Timing der Wartezeiten kümmern muss. Der Befehl WAITVID heißt ja nicht von ungefähr so: Er prüft zuerst, ob der letzte Schiebebefehl erledigt ist, und wartet ggf. solange.

Beispiel:

Code:

```
VSCL_preset     LONG  $1_004
VSCL_preset     LONG  %0_01_1_00_000_00000000000_0XX_0_11111111
                LONG $FF1F0700
                MOV VSCL, VSCL_preset
```

```
    WRVCFG, VCFG_preset
:loop
    WAITVID colors, #%%0123
    DJNZ #loop
```

Diesen Sägezahn (= 4 Bytes) können wir also schnellstmöglich alle 8 Takte ausgeben, macht 100 ns/4 Byte = 25 ns/Byte = 40 MB/sec Das ist nicht schlecht :-)

Wenn wir interessantere Signale ausgeben wollen, müssen wir sie natürlich entweder berechnen oder aus dem HUB-RAM holen, also z.B.

Code:

```
:loop
    RDLONG colors, adresse
    WAITVID colors, #%%0123
    ADD adresse, #4
    DJNZ periode, #loop
```

Das sind immer noch 10 MB/sec

Eine „handgemachte Schleife" ist nicht notwendigerweise viel langsamer:

Code:

```
:loop
    RDBYTE wert, adresse
    TEST wert , #iopinpos
    MOV OUTA, wert
    ADD adresse, #1
    DJNZ periode, #loop
```

Das sind etwa 3 MB/sec.

Wenn wir jetzt Video (4-6 MHz) oder VGA (25-30 MHz) betrachten, dann sehen wir aber, das diese Daten weder „zu Fuß" noch mit unserer „Teilnutzung" (1/4) des Videoschieberegisters erreicht werden können. Wir müssen das GANZE Schieberegister nutzen und kommen dadurch auf bequeme 40 MHz. Doch der Engpass ist nicht das Schieberegister, das mit durchaus höheren Clockwerten getaktet werden kann, sondern das Nachliefern der Daten aus dem HUB-Speicher!

Ein anderes Beispiel: Nur Nutzung eines einzigen I/O Pins statt 8.

Code:

```
VCFG_preset   LONG  $1_020
VSCL_preset   LONG  %0_01_0_00_000_00000000000_0XX_0_00000001
periode       LONG $0100
    WRVID VSCL, VSCL_preset
    WRVID VCFG, VCFG_preset

:loop
    RDLONG data32, adresse
    WAITVID colors, data32
    ADD adresse, #4
    DJNZ laenge, #loop
```

Wir können so 32 bit pro 400 ns ausgeben d.h. 80 Mbit/sek pro Kanal oder 40 Mbit/sek auf zwei Kanälen.

Eine Frage ist noch offen: Wie stellen wir eigentlich die Clock für das Schieberegister ein? Wie wir oben gesehen haben, können wir gut 80 MHz/Kanal gebrauchen. Für eine VGA Ausgabe bei der üblichen reduzierter Farbpalette (4 Farben je 32 pixel) etwa 30 MHz (bei 2x3 Farbkanälen). Das liegt im Rahmen der „normalen" Taktrate. Schnellere Taktraten können zu „bursts" und damit einem „jitter" führen, wenn das Schieberegister

nicht rechtzeitig „gefüttert" werden kann; in der Tat müssen wir den Takt so reduzieren, dass jeder WAITVID-Befehl noch einen kleinen Warteanteil hat.

Diese Einstellung wird mit dem **FRQA** Register gemacht, da der Timer-A für die Taktgenerierung des Videoshiftregisters zuständig ist.

Eine mögliche Einstellung ist z.B.

Code:

```
MOV    FRQA, #56          , 56*80/512 = ca 8,75 MHz
MOVI   CTRA, #%00001_101  'internal,  PLL = *16/4 = *4 = 35 MHz
```

Erklärung: Wenn – was wir nicht verhindern können im VGA/Video Mode! – wir die nachgeschaltete PLL verwenden, die durch die heruntergeteilte Clock aus dem Timer-A Überlauf angesteuert wird (und die 16-fache Frequenz zur Verfügung stellt), dann müssen wir gewisse Grenzen einhalten. Das Datenblatt sagt: Frequenz zwischen 4 und 8 MHz (16er-PLL = 64 bis 128 MHz). Das ist – bei PLLs aber nicht unüblich - ein ziemlich enger Bereich. Wir können hieraus dann aber bis zu %128 geteilte Frequenzen abgreifen, d.h. bis minimal 500 kHz.

Anders ausgedrückt: Die Clock, die wir mit dem Timer-A programmieren (die – laut Datenblatt eben zwischen 4 und 8 MHz liegen sollte), kann 16 fach erhöht oder 8 fach geteilt verwendet werden!

Zur Benutzung der Timer werde ich in Kürze im Assembler Teil einen kleinen Beitrag schreiben, aber es ist alles äußerst ausführlich in der Propeller-AN001 von Parallax erklärt!

Für „so ungefähr" werden oft nur die obersten 9 Bits von **FRQA** benutzt, die man mit dem MOVI Befehl gut setzen kann. Eine „Eins" dort führt also nach 512 Schritten zu einem Überlauf = 80 MHz/512 = 156 kHz (also viel zuwenig! 25 ist hier der Minimalwert!) Je größer der Wert, desto schneller, bei den im Beispiel angegebenen 56 also genau 8,75 MHz; es gibt hier natürlich einen Fehler, der sich im Prinzip in einem „Jitter" äußern würde. Wenn der Wert keine 2er-Potenz ist, wird der Puls einige Male etwas schneller erscheinen, um dann gelegentlich einmal auszufallen. Doch die nachgeschaltete PLL sorgt hier für einen Ausgleich!

So, bisher waren es alles unvollständige Beispiele. Ich gebe jetzt ein vollständiges Beispiel, das eine 8 I/O Gruppe - Pin 0.. 7 - einfach „hochzählt", und zwar so langsam wie möglich.

Code:

```
CON
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000

  _pinGroup = 0
  _pinMask = $FF<<(_pinGroup*8)

PUB main | n
  DIRA := _pinMask
  CTRA := %00001_000 << 23   ' internal, PLL % 128
  FRQA := POSX/CLKFREQ*4_000_000

  VSCL := $1_004       ' nur VIER Pixel ausgeben (Frame = 4)
  VCFG := %0_01_0_00_000_00000000000_000_0_11111111 + (_pinGroup <<
9) ' VGA Modus 2 Farben
  repeat
    WAITVID (n++ ,0)
```

Am MSB (Pin 7) messen wir etwa 240 Hz. Ich wollte eigentlich nur EIN PIXEL pro Frame ausgeben (VSCL := $1_001), aber das Signal sah da nicht stabil aus... Ist mir nicht klar, ob es da Einschränkungen bzgl der Framelänge gibt...

Doch es geht noch VIEL langsamer!
-1) Wir können die Pixelzeit strecken, bis zum 255 fachen

Code:

```
VSCL := $ff_0ff       ' nur EIN Pixel ausgeben, aber 255 fach
                      ' gestreckt
```

Am MSB messen wir jetzt 4Hz (=240/255*4)

-2) Wir können nun auch – ähnlich wie am Anfang 4 Bits - noch alle 32 gleichen (!) Bits im PIXELS Register rausschieben lassen; allerdings reicht das 12 Bit weite Frames-Feld im VSCL Register bei dieser Streckung nicht, wir können nur 16x verlängern:

Code:

```
VSCL := $ff_ff0       ' 16 mal ausgeben, aber 255 fach gestreckt
```

Am MSB messen wir jetzt ¼ Hz

-3) Zu guter letzt gucken wir noch mal, wie weit wir die PLL nach unten zum Fangen bringen können. Bei mir ging es bis 65 kHz (statt der spezifizierten 4 MHz)

Code:

```
CLKFREQ := 80MHZ/CLKFREQ*65_000
```

Am MSB messen wir jetzt gar nichts mehr Eine LED an Pin 0 jedoch blinkt brav im 2-Sekundenrhytmus vor sich hin (16*255*16/65_000 = 1 Sekunde) ...

Edit: Ich habe hier gerade eine SKizze zugefügt, die aber nicht gründlich überprüft ist; ev. gibt es noch die eine oder andere Vertauschung (VT/HT)

Nach oben

robbifan
Fortgeschrittener

Anmeldungsdatum: 24.05.2007
Beiträge: 84

Verfasst am: 02.12.2007, 13:21 Titel:

kann mir das tv-signal mal genau für anfänger erklären. mit einem einfachen spin-programm, wo man evtl nur ganz kleine asm-brocken reinbringt, wo es die spin-sprache evtl nicht schafft.

irgendwie wird hier rumgesprungen von a nach z und wieder nach j usw.
es ist da für einen anfänger keine klare programmstruktur zu erkennen.

mfg

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 02.12.2007, 17:49 Titel:

robbifan hat Folgendes geschrieben:

> kann mir das tv-signal mal genau für anfänger erklären. mit einem einfachen spin-programm, wo man evtl nur ganz kleine asm-brocken reinbringt, wo es die spin-sprache evtl nicht schafft.
>
> irgendwie wird hier rumgesprungen von a nach z und wieder nach j usw.

ist da für einen anfänger keine klare programmstruktur zu erkennen.

Oops, das trifft mich natürlich.....
Richtig ist, dass es einige "Brocken" waren, etwas aus dem Zusammenhang. Ich dachte jeder kann sich da was zusammenreimen:
-> Videosignal
-> Textreiber
-> Video Logik
Ich habe da kaum Maschinensprache verwendet...

O.k. ich bin wenig auf die PAL/NTSC -Farberzeugung eingegangen, was auch ein etwas aufwändigeres Kapitel ist....

Was erwartest Du denn, Robbifan? Es ist kein großer Akt, einen VGA Treiber zu schreiben (ca 1 Stunde), ein normgerechter TV-Treiber ist schwieriger wegen einiger Details des Interlacing.

Wesentlicher Aufwand entsteht durch den "work-around" Tile-Mapping und der Struktur des Zeichengenerators in ROM.

Das sind alles lästige Details, die schon ein paar Dutzend Seiten füllen können, wenn man sie nicht nur anreißen will.

Ich werde im Weihnachtsurlaub ein weiteres Kapitel hierzu in mein englisches Maschinensprachen-Tutorial aufnehmen.

Nach oben

robbifan
Fortgeschrittener

Anmeldungsdatum: 24.05.2007
Beiträge: 84

Verfasst am: 03.12.2007, 20:23 Titel:

du kannst doch ma' in spin einen code schreiben, der 1. nur eine linie darstellt. dann eine unterbrochene linie, dann ein quadrat ausgefüllt, dann das quadrat als ascii zeichen, bildschirm ganz füllen (alle scans darstellen).
zeitlich, wo das bild verzehrt wird usw. und nur ganz kleien brocken asm, wo spin nicht weiterkommt.

vga ist ja klar, alles läuft auf tv-hinaus.

Nach oben

deSilva
Ass

Anmeldungsdatum: 03.06.2007
Beiträge: 375

Verfasst am: 04.12.2007, 20:48 Titel:

VGA hat den Vorteil, dass der Code besser verständlich ist....

Wir müssen drei logische Schichten betrachten:
- die Videohardware, die mit einem LONG in WAITVID etwas ganz Bestimmtes macht
- die Schleife1, die eine Videozeile erzeugt
- die Schleife2, die eine Video(halb)bild aus Zeilen zusammensetzt.
- die Inhaltslogik, die dafür sorgt, dass in diesen Schleifen sinnvolle Information zur Verfügung steht.

Selbst die Schleife 2 kann man nicht in SPIN schreiben, da die Schleife 1 mit etwa 16kHz getaktet werden muss, wass für Spin normalerweise schon zuviel ist...

Der TV_Text.SPIN Treiber ist ein geniales Objekt, dass einen Bildwiederholspeicher füllt, der in einem Format gehalten ist, das von den verbreiteten TV- und VGA-Treibern gelesen wird.
....
Dein Vorschlag passt da irgendwie nicht so ganz zu...

Nach oben

Beiträge der letzten Zeit anzeigen: Alle Beiträge | Die ältesten zuerst | Los

neues Thema | Antwort erstellen | Parallax Propeller Forum Foren-Übersicht -> Allgemein | Alle Zeiten sind GMT + 1 Stunde

Seite 1 von 1

Gehe zu: Allgemein | Los

Du **kannst keine** Beiträge in dieses Forum schreiben.
Du **kannst** auf Beiträge in diesem Forum **nicht** antworten.
Du **kannst** deine Beiträge in diesem Forum **nicht** bearbeiten.
Du **kannst** deine Beiträge in diesem Forum **nicht** löschen.
Du **kannst** an Umfragen in diesem Forum **nicht** mitmachen.
Du **kannst** Dateien in diesem Forum **nicht** posten
Du **kannst** Dateien in diesem Forum **nicht** herunterladen