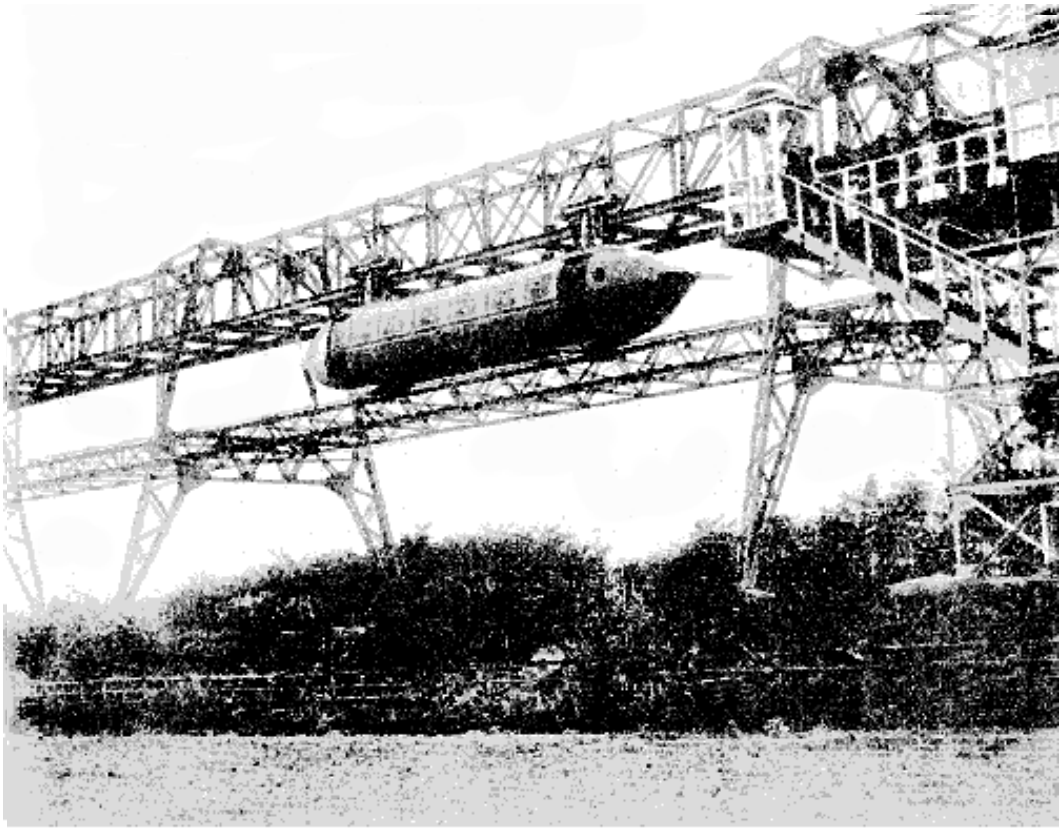
embedded

Rafael Deliano
Steinbergstr.37
82110 Germering
Tel 089/8418317
mail@embeddedFORTH.de

V1.0 (pdf) : 20. Mai 04

⑨



READ . ME

Die Portierung auf den 68HC908GP32 ist jetzt verfügbar, also wieder mehr Zeit für die Zeitschrift. Es ist nicht die beabsichtigte Zusammenstellung von Artikeln geworden. Aber es hat sich inzwischen soviel Material angesammelt daß es für eine weitere Ausgabe reicht. Vorgesehen für das nächste Heft ist u.a. das Filesystem für die CompactFlash-Karte.

Die Listings sind in nanoFORTH geschrieben. Für die Konvertierung in andere FORTH-Varianten sollte man im nanoFORTH-Manual nachlesen das jetzt in der F08-Version verfügbar ist.

Programmiergerät 68HC908

Motorolas FLASH-Controller haben eine weitgehend einheitliche serielle Schnittstelle die Programmierung und begrenzt Debugging von Software ermöglicht. Hier ist die Anschaltung mittels Einplatinencomputer dargestellt.

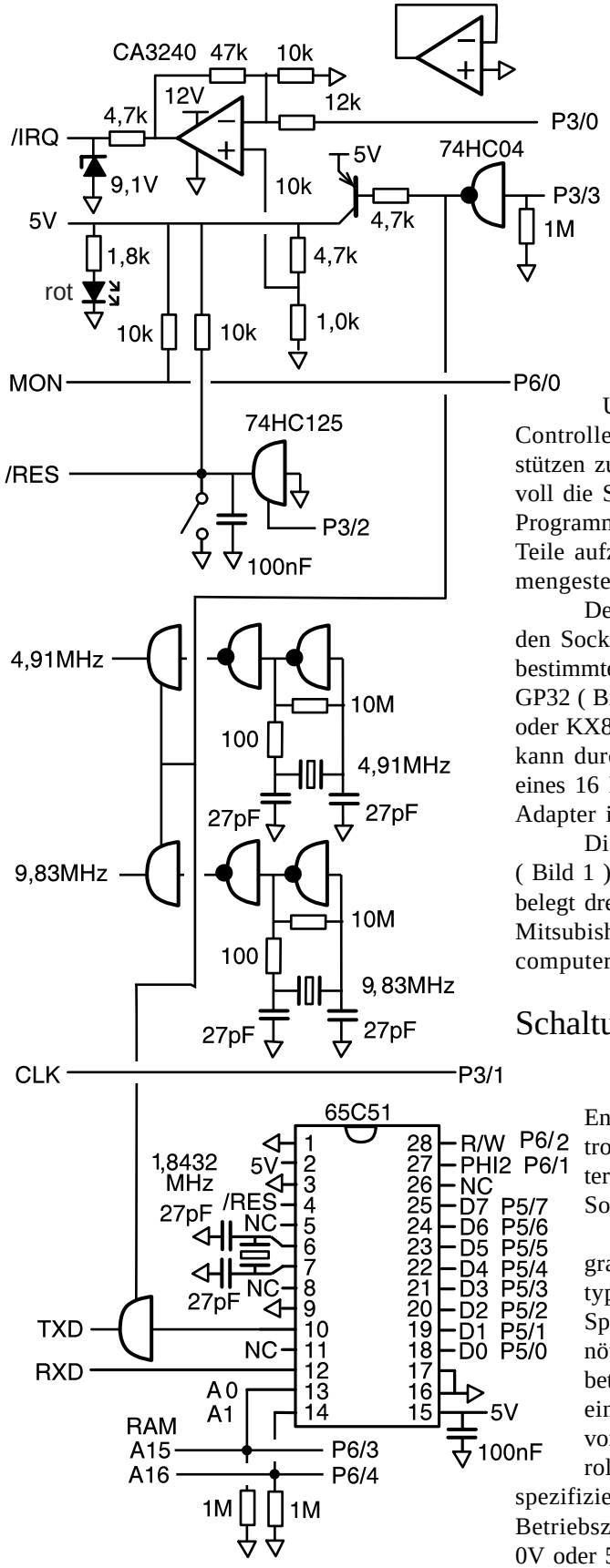


Bild 1:
Grundschaltung

Um verschiedene Controllervarianten unterstützen zu können ist es sinnvoll die Schaltung für das Programmiergerät in zwei Teile aufzuteilen die zusammengesteckt werden.

Der Adapter enthält den Sockel verdrahtet für ein bestimmtes Derivat. Z.B. GP32 (Bild 2), QT4 (Bild 3) oder KX8 (Bild 4). QY4 kann durch Parallelschaltung eines 16 Pin-Sockels in QT4-Adapter integriert werden.

Die Grundschaltung (Bild 1) ist universell und belegt drei Ports an einem Mitsubishi 6502 Einplatinencomputer.

Schaltung

Um Stecken und Entnehmen des Controllers zu erleichtern sind alle Pins per Software abschaltbar.

Für Programmiermodus ist typisch erhöhte Spannung am /IRQ-Pin nötig. Diese Spannung beträgt 7,5 ... 9V bei einer Stromaufnahme von 140uA die Motorola allerdings nicht spezifiziert. In anderen Betriebszuständen sind auch 0V oder 5V an diesem Pin erforderlich, sodaß er auf alle 3 Pegel einstellbar ausgeführt ist. Für die 0V ist ein CA3240 mit seiner CMOS-Eingangsgünstiger als ein LM358.

Die Versorgungsspannung 5V sollte speziell beim GP32 mit Maskenfehler (Code 3J20X) einen niederohmigen Entladewiderstand haben, hier 4,7k + 1,0k. Damit wird der 100nF Kerko im Adapter schnell auf unter 100mV entladen was für die Funktion des Power-On Resets in diesen Controllern wünschenswert ist.

Die Monitorschnittstelle besteht hier nur aus einem Portpin mit Pullup-Widerstand. Das Datenformat entspricht einer UART. Aber da das Protokoll halbduplex ist, kann diese leicht in Software nachgebildet werden.

Der Resetpin kann wahlweise manuell über Taster geschaltet werden, wofür der Kerko als Entprellung nötig ist. Der kleine Kerko auf dem GP32-Adapter verhindert Übersprechen vom 9,8MHz Takt und ist nur bei ungünstiger Leiterbahnführung erforderlich. Alternativ kann man Resetpuls per Software auslösen, wobei nach Hochschalten eine Verzögerng im Programm nötig ist, um die Zeitkonstante des Kerkos auszugleichen.

Während des Programmierbetriebs wird die CPU durch einen externen Takt versorgt. Die beiden üblichsten Frequenzen werden deshalb im Grundgerät erzeugt. Typisch wird 9,8MHz benötigt.

Einige Controller haben intern als RC-Generatoren statt externe Quarze. Diese sind per FLASH abgleichbar. Den Abgleich kann das Programmiergerät machen. Zur Ausgabe des Takts für Messung ist es nützlich vom Adapter an einen Pin des Mitsubishi eine Leitung zu führen, die hier als CLK bezeichnet ist.

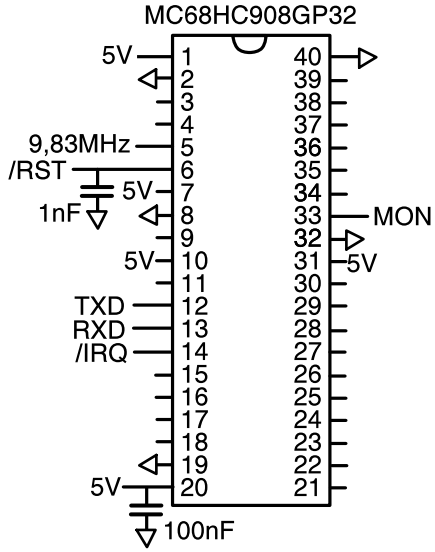


Bild 2: Adapter GP32

Größere Varianten des 68HC908 wie der GP32 haben eine UART eingebaut. Eine weitere serielle Schnittstelle zum Controller ist für ein Programmiergerät nicht notwendig. Aber z.B. für Tests von Software doch oft nützlich. Hier wurde deshalb eine 65C51-UART an die Ports gehängt. Dieser Typ ist inzwischen obsolet, ein 16C550 würde die gleiche Funktion erfüllen.

Da der GP32 ein 32k Byte FLASH hat wurde es auch nötig den Speicherbereich des Mitsubishi durch ein gebanktes 128k SRAM zu erweitern. Die Portpins für die Steuerung der Banks werden von der 65C51 UART mitverwendet.

Erweiterungen

Die in Bild 1 dargestellte Schnittstelle ist ein Kompromiß aus Flexibilität und Aufwand. Einige Features fehlen noch.

Für Controller mit internem RC-Generator wäre es günstig, auch die Versorgungsspannung des Controllers per D/A-Wandler des Mitsubishi von 2,7V - 5,5V einstellbar zu machen. Denn dieser Takt ändert sich etwas mit der Spannung. Man könnte damit noch genauer abgleichen.

Erste Voraussetzung dafür ist eine Umstellung der 1 Pin Monitorschnittstelle auf 2 Pins RXD und TXD, sodaß man die Pegelumsetzung leichter durchführen kann. Die müssen dann natürlich auch an allen anderen Schnittstellenpins eingebaut werden.

Speziell bei Controllern in SMD-Bauform ist es angebracht diese erst einzulöten und dann zu programmieren. Kontaktierung erfolgt dabei typisch über Nadeladapter. Dabei sind weitere Guardschaltungen nötig, damit die Beschaltung im Zielsystem geeignet neutralisiert wird. Auch wenn die Beschaltung dafür im Adapter vorgenommen wird, sind meist erhebliche Anpassungen in der Grundschaltung nötig.

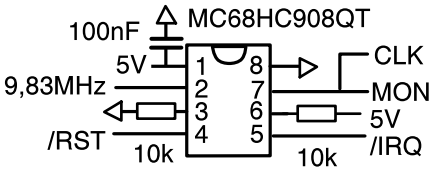


Bild 3: Adapter QT1 bzw. QT4

Monitor

Die Beschreibung der Schnittstelle findet sich in den Motorola-Unterlagen. Hier ist deshalb nur dargestellt wie sie in der Software abgebildet wurde.

Die Schnittstelle ist zwar halbduplex auf einem Pin, entspricht aber sonst nominell dem Format einer UART mit 9600 Baud 8N1, bei einer Busfrequenz des Controllers von 2,45 MHz. Der 68HC908 macht sie in Software, weshalb diverse Pausen eintreten, in denen er die Schnittstelle nicht bedient. Es hat sich deshalb herausgestellt, daß es auch im Host sinnvoll ist die Schnittstelle per Software zu emulieren. Echte UART macht nur Probleme.

Die unterste Ebene der UART-Funktionen:

```
(EMIT) \ in: akku
(KEY)  \ out: akku
BREAK? \ ( – F1 ) 1 = error
BREAK! \ (  –  )
```

Neben dem Lesen und Schreiben von Bytes wird auch Erzeugen und Erkennen von Break-Signalen, also 11 Bits low, benötigt (Listing UART0).

Im Protokoll kommt jedes Byte das der Host schickt vom Controller als Echo zurück.

```
C!^? \ ( C – F ) F=1 error
C!^  \ ( C –  )
!^   \ ( N –  )
C@^  \ (  – C )
@^   \ (  –  N )
```

Die nächste Befehlsschicht führt das Lesen und Schreiben von Bytes und 16 Bit Worten (für Adressen) zusammen mit den nötigen Verzögerungen durch.

Die oberste Schicht bildet den Befehlsumfang den das Monitor-ROM des Controllers hat direkt ab (Listing UART1). Sobald die Monitorverbindung aufgebaut ist, kann man diese Befehle interaktiv von FORTH aus benutzen.

Grundlegend sind `TC@` (Motorola: „READ“) Byte lesen und `TC!` („WRITE“) Byte schreiben im 64k Adreßraum. Ergänzend dazu gibt es „indexed“ Varianten `iTC@` („IREAD“) und `iTC!` („IWRITE“) bei denen die Adresse automatisch inkrementiert wird und deshalb nicht übertragen werden muß. `iTC@` hat die Eigenheit beide folgenden Bytes auszulesen.

```
TC@     \ ( adr – C1 )
TC!     \ ( C1 adr –  )
iTC@    \ (  – C1 C2 )
iTC!    \ ( C1 –  )
READSP  \ (  – addr )
RUN     \ (  –  )
```

Um sich den Speicher als Hexdump auf den Bildschirm zu holen ist nützlich:

```
TDUMP     \ ( addr – )
```

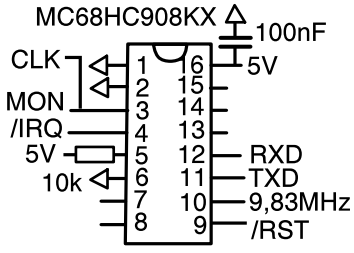


Bild 4: Adapter KX8

Programmausführung RUN („RUN“) wird über Rücksprung aus IRQ-Unterprogramm realisiert. Beim Einstieg in Monitormodus wurde von der CPU ein PSH, und SWI, Opcode ausgeführt. Ersterer legt das obere Byte des 16 Bit X-Registers auf den Stack. SWI ist ein per Opcode ausgelöster Interrupt.

Wo Stackpointer gerade hinzeigt kann man durch den Befehl READSP („READSP“) auslesen. Man kann den Stack also passend mit TC! doktern bevor man mit RUN den Rücksprung auf neue Adresse ausführt.

Soweit Testprogramme die man ins RAM des Controllers geladen hat mit dem RTI,-Opcode enden, kehren sie in den Monitormodus zurück. Es ist aber dann noch ein Break nötig um die Schnittstelle wieder zu aktivieren.

Monitor Entry

Mit dem Befehl UP schaltet man am Programmieradapter die Versorgung des Controller ein. Mit dem Befehl DOWN schaltet man sie aus.

Hat man die Versorgung hochgeschaltet, muß der Host erst 8 Bytes Sicherheitscode schicken, die mit FLASH verglichen werden. Bei fabrikneuem Controller ist der Inhalt normalerweise FF. Aber die Bytes des Sicherheitscodes sind auch die Interruptvektoren, Also nach Programmierung einer Applikation meist zwangsläufig verändert. Die 8 Referenzbytes werden deshalb aus der Variable SEC-CODE entnommen die auf FF initialisiert wurde. Verändert man die Bytes im Controller muß man die Bytes in der Variable anpassen.

Bei Eingabe eines falschen Codes geht der Controller in einen eingeschränkten Monitormodus. Von dort kann man das FLASH zumindest komplett löschen. Ob die Initialisierung korrekt war, kann man durch Lesen eines Bytes im RAM feststellen. Der Befehl UP tut das und druckt eine Warnung wenn man Initialisierung nicht voll geschafft hat (Listing PROG 2).

Tabelle 1: Anwenderbefehle

```
PROGRAM   \ ( – F ) F = 1 error  copy RAM to FLASH
ERASE     \ ( – F ) F = 1 error  erase FLASH
READ      \ ( – F ) F = 1 error  copy FLASH to RAM
VERIFY    \ ( – F ) F = 1 error  compare to FLASH
ERASED?   \ ( – F ) F = 1 error  check if FLASH = FF
```

Programmierung

Die üblichen Funktionen, die der Benutzer haben will, kann man direkt auf FORTH-Befehle abbilden (Tabelle 1).

VERIFY und ERASED? kann man auf READ zurückführen, wenn diese Funktion einen eigenen Zwischenspeicher im RAM hat. Man liest dann erst mit READ komplett aus und führt VERIFY und ERASED? bequem gegen Daten im RAM durch. Dadurch umgeht man insbesondere Probleme die sich aus der Fragmentierung des Speichers ergeben.

Das FLASH ist nominell in einzelne Pages organisiert. Diese sind bei großen Controllern 64d, bei kleinen 32d Bytes lang. Man kann das FLASH entweder komplett löschen („bulk-erase“), oder nur eine einzelne Page. Letzteres wird hier nicht benutzt. Schreiben von FLASH ist byteweise möglich. Aber nicht so einfach wie z.B. bei EEPROM, sondern mit Routinen die auf Pages ausgerichtet sind.

Für große Controller wie den GP32 müssen alle FLASH-Grundfunktionen durch Programme die man ins RAM des Controllers kopiert und dort ausführt realisiert werden. Kleine Varianten wie der QT4 haben nicht genügend RAM, Motorola hat deshalb zusätzlich ein ROM mit Unterprogrammen eingebaut. Die Programme im RAM werden dadurch genügend klein. Man hat aber nun die Programmstruktur etwas starr vorgegeben. Um die Software für das Programmiergerät einheitlich zu halten, wurde deshalb anhand des QT4 implementiert und dann auf GP32 portiert.

ROM-Routinen für QT4 & KX8

Die Unterprogramme des zusätzlichen ROMs sind zwar identisch ausgeführt, die Adressen für Speicherbereiche, Variablen und Einsprungpunkte variieren aber (Tabelle 2).

Daten belegen dabei im RAM den Bereich ab DATA mit bis zu 32d Byte Länge, d.h. einer Page des FLASH. Der Endwert wird in der Variable LADDR vorgegeben, während der Startwert FADDR bei Aufruf der entsprechenden Routinen in Akku und X-Register übergeben wird. Beide Werte müssen nicht auf Page-Grenzen abgestimmt sein.

GETBYTE

Dieses Unterprogramm liest ein Byte über die Monitorschnittstelle ein. Auf Echo des Bytes wird hier aber verzichted um die Geschwindigkeit der Datenübertragung zu erhöhen. Beispiel:

```
   DDRA 0 MBC, \ inputPA0
GETBYTE  JSR,
```

RDVRRNG

Dieses Programm dient zum Lesen bzw Verifizieren von FLASH.

Wenn beim Aufruf der Akku Null ist werden die Daten auf der Monitor-Schnittstelle ausgegeben. Wenn bei Aufruf der Akku ungleich Null ist, werden die Daten in DATA abgespeichert, nachdem sie vorher mit den dort liegenden Daten verglichen wurden. Ergebnis des Vergleichs enthält das Carry Flag. Ist es gesetzt ist, war der Vergleich ok. Ferner enthält der Akku die Prüf-

Tabelle 2: Adressen

	QT4	KX8
RAM		
Start	80	40
CTRLBYT	88	48
CPUSPD	89	49
LADDR	8A	4A
1+	8B	4B
DATA	8C	4C
max	AB	6B
I/O		
Monitor-Pin	PA0	PA0
FLASH-ROM		
GETBYTE	2800	1000
RDVRRNG	2803	1003
ERARNGE	2806	1006
PRGRNGE	2809	1009
DELNUS	280C	100C

summe über die gelesenen Daten. Beispiel um von F000 bis F010 auf Monitorschnittstelle auszugeben:

```
      A. CLR,
F010 #. LDHX,
LADDR   STHX,
F000 #. LDHX,
RDVRRNG  JSR,
```

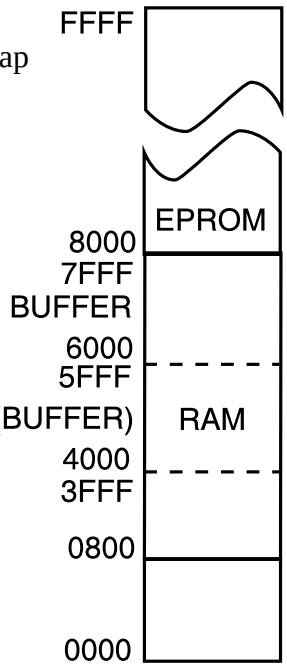
ERARNGE

Löscht FLASH pageweise bzw. den kompletten Speicher.

Wenn Bit 6 in Variable `CTRLBYT` gesetzt ist wird der komplette Speicher gelöscht. Andernfalls nur eine Page (32 Byte). Die Page wird durch Startwert spezifiziert und ist an Pagegrenzen ausgerichtet. Da fürs Timing die Verzögerungsschleife `DELNUS` verwendet wird, muß für diese die Variable `CPUSPD` initialisiert sein.

Diese Routine war in den ersten Versionen des QT4- bzw., KX8-Controller teilweise defekt. Es wird dann irrtümlich der die Page der Vektoren gelöscht. Um den kompletten Speicher zu löschen jedoch auch dort verwendbar. Beispiel um mit 8 MHz (8x4=20h) Taktfrequenz kompletten Speicher zu löschen:

Bild 5: MemoryMap Host 32kSRAM für Targets bis 8k Byte



Stack used:
4 byte
6
5
7
3

```
CPUSPD 20 #. MOV,
CTRLBYT    6 MBS,
F000     #. LDHX, \ dummy
ERARNGE     JSR,
```

PRGRNGE

Bytes von `DATA` in FLASH schreiben. Dabei wird angenommen, daß das FLASH bereits gelöscht ist. Es wird auch kein Verify durchgeführt. Da fürs Timing die Verzögerungsschleife `DELNUS` verwendet wird, muß für diese `CPUSPD` initialisiert sein.

DELNUS

Die Verzögerungsschleife wird durch die Übergabe von Werten in Akku und X-Register programmiert. Dabei spezifiziert der Akku die Busfrequenz der CPU (Tabelle 3) von z.B. 1 MHz. Bei diesem Wert wäre der Verzögerungswert in X von 1 - 255 * 12usec einstellbar. Wenn der Takt anders als 1 MHz ist, muß man die 12usec auch passend skalieren.
`ERARNGE` und `PRGRNGE` liefern den Wert für Akku selbst, aber die Busfreqenz muß man in der Variablen `CPUSPD` vorher initialisieren. Für die Busfrequenz 2,45 MHz also mit 0Ah. Für 8 MHz mit 20h.

Tabelle 3: Taktanpassung

Akku =	Operating Frequency
0	illegal
1	0,25 MHz
2	0,50 MHz
3	0,75 MHz
4	1,00 MHz
usw.	

FLASHer für QT4

Da für jede Controller-Variante eine Menge Adressen neu zu definieren ist, werden diese beim Compilieren per File festgelegt (Listing PROG2). Dort würde man auch Anpassung an Variante QT1 mit kleinerem Speicher machen.

Um Datenblöcke und Programme ins RAM des Controller zu übertragen und dann auszuführen, gibt es einheitliche Unterprogramme:

```
TCOPY \ ( C1 – )
    \ C1 = number of bytes
DCOPY \ (  –  )
     \Page of data 32 bytes
TEXECUTE  \ (  –  )
```

Da das längste Programm nur 70 Bytes lang ist, paßt es bequem in die Zeropage. Im Host liegt der 8k Hauptspeicher `BUFFER` von 6000 ... 7FFF. Hier wird das Programm für den Controller per Cross-Compiler oder Intel-HEX-Loader abgelegt. `(READ)` hat seinen Zwischenspeicher `(BUFFER)` von 4000 ... 5FFF

FLASH-Programmierung

Das FLASH ist leider völlig fragmentiert. Es gibt einzelne Bytes die man programmieren muß. Kleine Blöcke wie die Vektoren. Und den umfangreichen Hauptspeicher.
Aus diesem Grund sind zwei Treiber nötig: `(32PROG)` und `(STREAM-PROG)`. Beide führen auch Verify durch und geben entsprechendes Flag zurück.
`(32PROG)` erledigt einzelne Bytes und 32 Byte Pages oder Abschnitte

innerhalb einer Page. Datenübertragung erfolgt langsam mit DCOPY (Listing PROG4)

(STREAM-PROG) akzeptiert für Anfang und Ende nur auf 256 Byte Blöcke ausgerichtet Grenzen. Erledigt den Hauptspeicher aber in einem Durchlauf. Dazu empfängt das geladene Programme über die Monitorschnittstelle Pages zu 32 Bytes und programmiert sie. Dann gibt es kurz einen kurzen Lowpuls auf der Schnittstelle aus, um die nächste Page anzufordern (Listing PROG5).

Mit diesen beiden Grundbefehlen verarbeitet der Befehl PROGRAM alle Teilbereiche (Listing PROG6). Das Schreibschutzregister FLBPR kann danach durch den Befehl PROG-FLBPR geschrieben werden.

Wenn der Speicher des Controllers nicht komplett durch ein Programm belegt ist, kann man die Programmierzeit für den Hauptspeicher verkürzen, indem man nur bis zur Programmobergrenze programmiert. Dazu muß der Buffer vor dem Compilieren auf FF initialisiert sein, damit man die Grenze feststellen kann.

Erase

Im QT4 denkbar harmlos, weil man nur die Routine im ROM aufrufen muß (Listing PROG7)

Read

Auch hier ermöglicht es (READ) eine Routine im ROM vom Start des Hauptspeicher bis FFFF in einem Zug die Bytes herauszulesen und im Zwischenspeicher abzulegen. ERASED? vergleicht den Inhalt des Zwischenspeichers mit FF, während VERIFY den Vergleich gegen den Hauptspeicher durchführt. READ kopiert hingegen den Zwischenspeicher in den Hauptspeicher.

FLASHer für GP32

Dieser Controller hat viel RAM und deshalb kein zusätzliches ROM. Die Programme die man deshalb ins RAM lädt werden entsprechend länger, hier bis zu 200 Bytes. In der Zeropage kommen deshalb nur die Daten unter. Programme werden ab 0100h gespeichert. Die FLASH-Page dieses Controllers ist 64 Bytes. Ansonsten ist die Software ähnlich wie für den QT4 aufgebaut.

ERASE hat die Besonderheit, daß man eine Adresse in der Software ändern muß, wenn mit falschem Security String initialisiert wurde. Das wird aber über ein Flag das von UP gesetzt wird automatisch gesteuert.

Da dieser Controller 32k Byte FLASH hat, sind die Speicherprobleme im Host gravierend, wenn man eine 8 Bit CPU verwendet. Hier wurde das 32k SRAM deshalb durch ein gebanktes 128k SRAM ersetzt (Bild 6). Der Hauptspeicher BUFFER in BANK = 0 ist trotzdem auf 16k Byte begrenzt. Die verdeckten Speicherseiten BANK 1, 2, 3 sind umständlich anzusprechen und werden deshalb nur von (READ) verwendet (Tabelle 4).

ERASED? kann unschwer den gesamten Speocher prüfen. PROGRAM und VERIFY sind jedoch auf 16k Byte beschränkt. Dieses wird durch die Variable MAP eingestellt (Tabelle 5).

Die (un)freiwillige Beschränkung auf 16k Byte hat ihre Ursache in der Geschwindigkeit: um 16k mit 9600 Baud zu übertragen, braucht man bereits 17sec. Programmierzeit im Controller und allgemeiner overhead dehnen das nochmal deutlich.

Man muß sich für Controller mit grossem Speicher also von der Kompatibilität zum QT4 beim internen Aufbaus der Software lösen und vor allem die Datenübertragung beschleunigen. Wenn man den Controller auf Programmiergerät in Sockel steckt, kann man die Bytes z.B. auch parallel über Portpins einlesen. Auch schnellere serielle Verfahren für in-circuit Programmierung auf der Leiterplatte sind denkbar.

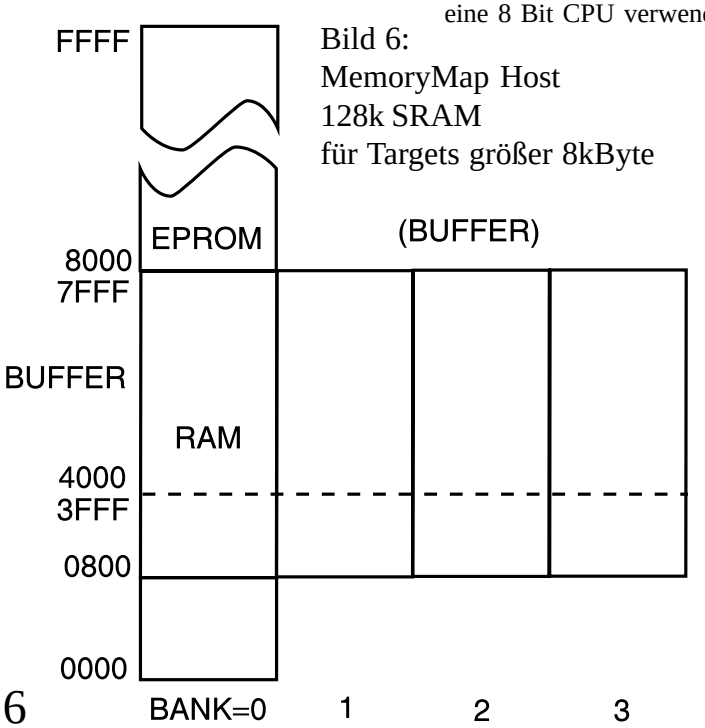


Bild 6:
MemoryMap Host
128k SRAM
für Targets größer 8kByte

Tabelle 4: BANK-Zuordung für (READ)

Host	Target
(BANK=1	4000 ... 7FFF 4000 ... 7FFF)
BANK=2	4000 ... 7FFF 8000 ... CFFF
BANK=3	4000 ... 7FFF D000 ... FFFF

Tabelle 5: MAP-Zuordnung für PROGRAM, VERIFY

Host BANK=0	Target
MAP=0	illegal
MAP=1	4000 ... 7FFF 4000 ... 7FFF
MAP=2	4000 ... 7FFF 8000 ... CFFF
MAP=3	4000 ... 7FFF D000 ... FFFF

Frequenzkalibrierung QT4 & KX8

Der 12,8 MHz RC-Oszillator der Q-Typen hat unabgeglichen nur eine Genauigkeit von +/-25%. Motorola liefert zwar in einem FLASH-Register einen Korrekturwert der ihn auf +/-5% bringt. Löscht man das FLASH jedoch via Bulk-Erase, geht auch der Korrekturwert verloren. Die eigentliche Steuerung des Oszillators erfolgt dabei durch das I/O-Register

```
0038 OSCTRIM
```

das durch Reset auf 80h eingestellt wird. Der FLASH-Korrekturwert liegt in

```
FFC0 TRIMLOC
```

und enthält nach Mass-Erase FF. Für die Initialisierung nach Reset ist der Anwender zuständig. Z.B. nach Reset so:

```
FFC0 LDA, 0038 STA,
```

Signal

Im Monitor-Betrieb mit hoher Spannung am /IRQ-Pin wird der Controller durch den externen Takt gespeist. Schaltet man die Spannung am /IRQ-Pin auf 5V, wird auf internen Oszillator umgeschaltet. Gleichzeitig wird auch der Watchdog aktiviert. Dieser löst nach 81,9msec aus.

Man lädt also erst in den Controller ein Programm das eine 100Hz Frequenz am Monitorpin erzeugt und gleichzeitig den Watchdog bedient. Wenn dieses läuft senkt man die Spannung an /IRQ ab. Damit wird die Busfrequenz von quarzgenau 2,45 MHz auf nun nominell 3,2 MHz umgeschaltet und das Rechtecksignal ändert sich entsprechend.

Tabelle 7: nomineller Einstellbereich

```
OSCTRIM 00h = -25% = 0,75
         80h =   0% = 1,00
         FFh = +25% = 1,25
```

Bild 7: Formel zu Berechnung des Korrekturwerts

```
(512+128) - ( T80h  * 512 / Tsoll ) = OSCTRIM
```

Tabelle 6: Meßwerte eines Musters

```
  1451 * 8,98usec  = 13,03msec  @ 2,45 MHz Quarz
( 1111               9,97          Soll 3,2 MHz )
  1368              12,28      @ OSCTRIM = 80h
  1081               9,71                  00h
  1638              14,71                  FFh
```

Timer

Auf dem Mitsubishi-Einplatinencomputer ist es am einfachsten den 10msec Lowpuls in einer Zählschleife in Software auszulesen. Dieser 16 Bit Timer hat eine Auflösung von 8,98usec. Die genauen Werte der Schleifen in beiden CPUs sind nicht so kritisch, da man ja mit den 2,45 MHz einen Kalibrierlauf machen kann um den Sollwert zu bestimmen. Tabelle 6 zeigt Meßwewerte von einem Controller.

Wie man dort auch sieht deckt der Korrekturbereich von OSCTRIM typisch +/-25% ab (Tabelle 7) und kann als näherungsweise linear angenommen werden.

Nach einer Messung mit `OSCTRIM` = 80h ist es deshalb möglich anhand des Messwerts und des Sollwerts Tsoll (hier 1111) einen neuen Korrekturwert für `OSCTRIM` ausrechnen der schon ziemlich genau ist (Bild 7). Bei der Berechnung muß man allerdings zusätzlich Schranken setzen (`T-LIMIT-`, `T-LIMIT+` Tabelle 8), damit man den Ergebnisbereich 00 ... FF bei ungünstigen Eingangswerten nicht überschreitet.

Da man aber nicht sicherstellen kann, daß die gewünschte Toleranz unmittelbar getroffen wird, ist die verwendete Kalibrierroutine (Listing CALIB) zweistufig. Nach dem Grobschritt mit der Formel wird in bis zu 10 Schritten noch um +1 bzw. -1 geändert um das Toleranzband zu erreichen. Da jeder Schritt etwa 0,2% entspricht, sollte eine Frequenz +/-0,5% leicht erreichbar sein (Tabelle 8).

Den endgültig bestimmten Wert wird man im Hauptspeicher bei `TRIMLOC` ablegen und dann alles zusammen mit `PROGRAM` ins FLASH kopieren. Alternativ kann man mit `(32PROG)` auch eine Routine schreiben die nur das einzelne Byte programmiert.

KX8

Die Frequenz wird durch eine PLL aus einem internen 307,2kHz RC-Oszillator erzeugt. Dieser schwankt durch Bauteilstreuung um +/-25%, kann aber durch durch Laden eines Korrekturwerts in

```
0038  ICGTR
```

abgeglichen werden. Nach Reset ist 80h geladen. Der Takt ist in Schritten von 0,2% abstimmbar.

Der Prozessor hätte im ROM Trimroutinen [3], die hier aber nicht verwendet werden. Die Software für den QT4 ist nämlich direkt verwendbar. Auch hier wird nach Absenken von /IRQ von 9V auf 5V vom externen Takt auf den internen umgeschaltet. Da der Frequenzmultiplizierer bei Reset auf 21d initialisiert wird, ergibt sich ein 6,45 MHz Takt. Daraus die Busfrequenz 1,613 MHz. Die geänderten Konstanten zeigt Tabelle 9

[1] Motorola AN1831 „Using C68HC908 ON-Chip FLASH Programming Routines“

[2] AN2312 „MC68HC908QY4 Internal Oszillator Usage Notes“

[3] AN1831 „Using MC68HC908 On-Chip FLASH Programming Routines“

Tabelle 8: Konstanten QT4 3,2 MHz

```
D%  836 CONSTANT T-LIMIT-
D% 1105 CONSTANT T-SOLL-% \ +0,5%
D% 1111 CONSTANT T-SOLL
D% 1117 CONSTANT T-SOLL+% \ -0,5%
D% 1389 CONSTANT T-LIMIT+
```

Tabelle 9: KX8 1,613 MHz

```
D% 1658 CONSTANT T-LIMIT-
D% 2192 CONSTANT T-SOLL-% \ +0,5%
D% 2204 CONSTANT T-SOLL
D% 2215 CONSTANT T-SOLL+% \ -0,5%
D% 2759 CONSTANT T-LIMIT+
```

OCR-Eingabegerät

Eigenbau ist wegen der benötigten Mechanik und Optik mühsam. Alternative ist die „Pistole“ (Bild 1) eines Optoline 3130 bzw 3140 der CGK Konstanz. Die Geräte wurden ab ca. 1983 hergestellt und sind heute oft ausgemustert preiswert erhältlich.

Bild 1: Lesegerät

Im Kopf (Bild 2) sind steckbar zwei Glühlampen angebracht die mit regelbarer Lichtstärke das Papier beleuchten. Hinter einer etwa 4cm langen Optik befindet sich ein CCD-Bildaufnehmer mit 16x64 Pixel Auflösung. Aus der Optik wird über eine BPW34 Fotodiode die Lichtstärke ausgekoppelt und ins Steuergerät übertragen. Das grüne LED dient als Rückmeldung für den Benutzer, seine Steuerleitung wird aber auch in der Schaltung verwendet.

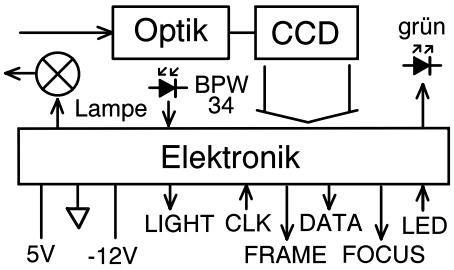


Bild 2: Blockschaltbild

Verbindung zum Hauptgerät erfolgt über einen 15pol SubD-Stecker auf dem alle Signale Logikpegel haben (Tabelle 1). Die Stromaufnahme der 5V variiert stark mit der Beleuchtung durch die Glühlampen. Das Signal an Pin 6 zeigt über das in die CCD reflektierte Licht an ob die Pistole Papier im Blickfeld hat. Die eigentlichen Daten kommen mit TTL-Logikpegel an Pin 7 und der Rahmentakt an Pin 12. Takt wird vom Steuergerät an Pin 11 dazu geliefert.

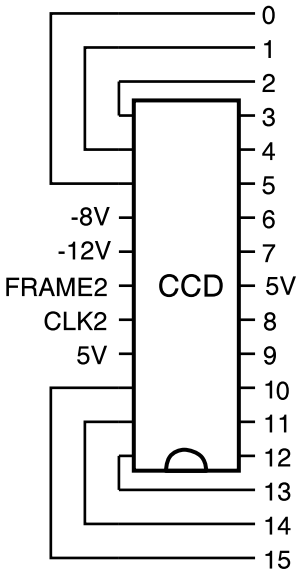
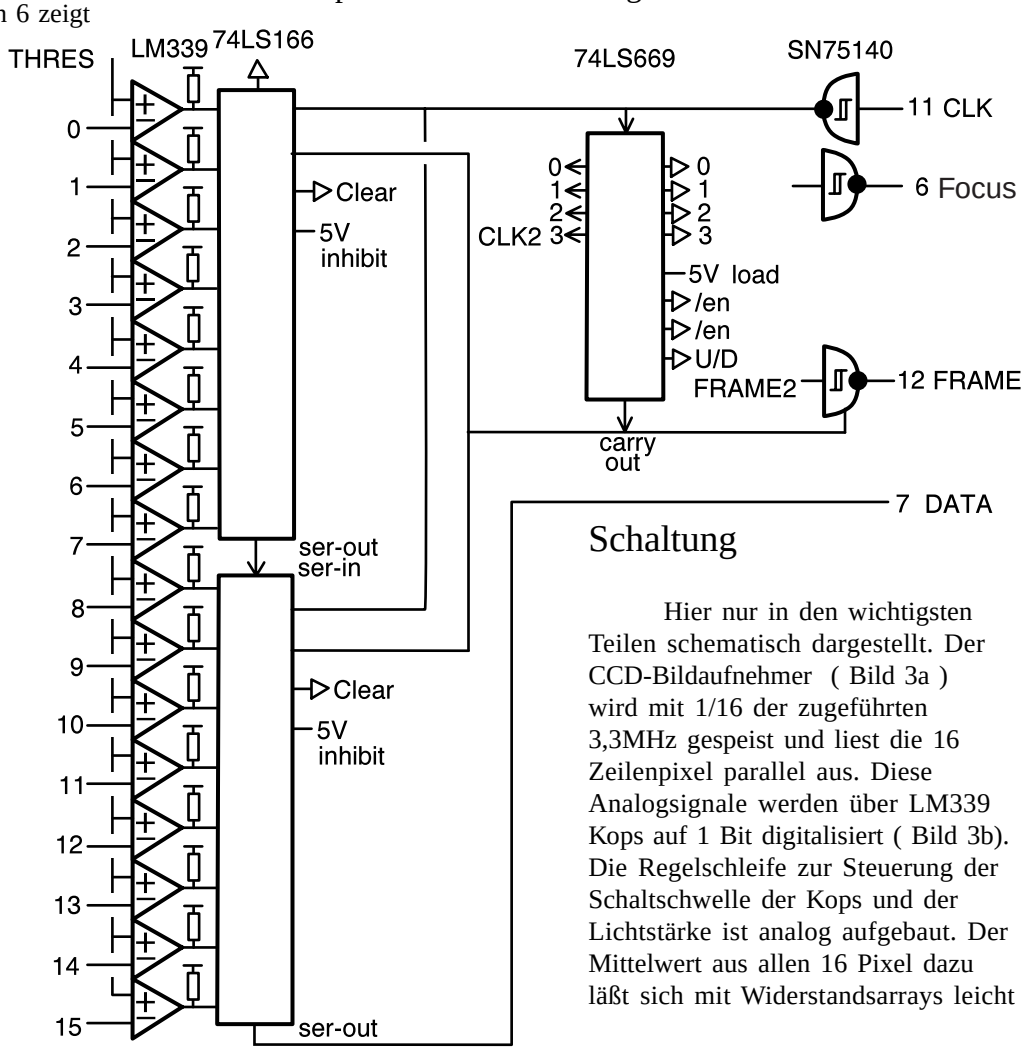


Bild3a: CCD

Bild 3b: Komparatoren & Schieberegister



Schaltung

Hier nur in den wichtigsten Teilen schematisch dargestellt. Der CCD-Bildaufnehmer (Bild 3a) wird mit 1/16 der zugeführten 3,3MHz gespeist und liest die 16 Zeilenpixel parallel aus. Diese Analogsignale werden über LM339 Kops auf 1 Bit digitalisiert (Bild 3b). Die Regelschleife zur Steuerung der Schaltschwelle der Kops und der Lichtstärke ist analog aufgebaut. Der Mittelwert aus allen 16 Pixel dazu läßt sich mit Widerstandsarrays leicht

Tabelle 1: Schnittstelle am orginalen Steuergerät

```
1   GND
2   GND
3   5V    320mA max.
4   5V
5   -12V   150mA
6   out FOCUS ( SN75140 )
    1 = out of focus
    0 = in focus
7   out DATA  ( 74LS166 )
8   NC
    vom Grundgerät low
    gesendet
9   NC
     vom Grundgerät mit
     10k auf 5V gezogen
10  in LED
     über 120 Ohm an 5V
     0 = LED on   ( 20mA )
     1 = LED off
     im Grundgerät 350usec
     Takt mit etwas über
     50% Taktverhältnis
11  in CLK  ( 75140 )
    3,3 MHz
12  out FRAME  ( 75140 )
     Rahmensignal: ca.
     320usec low dann ca.
     5usec high
13  NC
14  out LIGHT
    open collector mit
    1,5k pullup
     Lowpegel: -0,8V
15  NC
```

bestimmen (Bild 3c) . Das digitale 16 Bit Datenwort wird mit den beiden 74LS166 Schieberegistern auf seriell gewandelt und aufs Kabel ausgegeben (Bild 3b). Der Rahmentakt FRAME wird in der CCD erzeugt und pulst während der obersten Zeile high (Bild 4).

Adapter

Als Steuergerät wird hier ein 8 Bit Einplatinencomputer basierend auf Mitsubishi-6502 mit 32kByte RAM verwendet. Der kann natürlich nicht den kontinuierlichen Datenstrom aufnehmen, aber einzelne Bilder.

Die seriellen Daten vom Kabel werden über Schieberegister wieder in in ein 16 Bit Datenwort gewandelt (Bild 5). Mit dem Orginaltakt 3,3 MHz kommen die Datenworte ca. alle 5usec. Das ist annähernd durch linearen Code [1] zu bewältigen. Dazu muß aber das Lesegerät mit dem Takt des Prozessors gespeist werden. Durch die Laufzeit der Opcodes hat man starres Teilerverhältnis. Hier ergibt sich z.B. daß die Adressen der Schieberegister in der Zeropage als auf Ports liegen müssen, damit jeder Zugriff 16 Zyklen dauert. Das ist ein „gerader“ Wert der mit der synchron laufenden externen Logik nachgebildet werden kann. Da aber keine zwei Ports komplett frei sind, müssen hier die beiden Pins der UART über einen 74HC4053 während des Einlesens eines Bildes abgetrennt werden.

Mit den 9,84 MHz des Controllers ergeben sich an der Schnittstelle so 2,45 MHz, also 25% weniger als im Orginal. Verdaut das Lesegerät aber recht gut. Man könnte den Quarz des Controllers auf 12,28 MHz erhöhen, kommt dann auf 3,07 MHz also -7%. Allerdings muß man dann die Einstellung der Baudrate in der UART in der FORTH-Firmware patchen.

Da das Kabel 2m lang ist, sind an DATA, FRAME, CLK passende Terminierungen nötig. Trotzdem sind die Pegel nicht optimal weil 74HC und TTL-Logik gemischt sind.

Die Schaltschwelle der Kops wird durch das Signal LED vom Grundgerät mitbeeinflußt. Es hat am Orginalgerät die Periode eines Frames und 50% Tastverhältnis. Dabei wird der Lowpuls konstanter Länge in der Mitte innerhalb des Frames hin- und hergeschoben. Die Nachbildung hier ist stark vereinfacht, es erfolgt manuell eine feste Einstellung durch Poti. Das hat sich bei Vorlagen mit gutem Kontrast als ausreichend erwiesen.

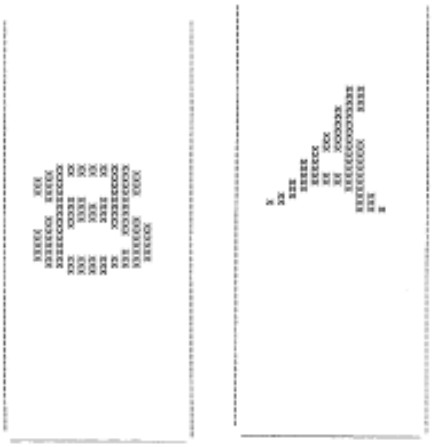


Bild 6 : Ausdruck als ASCII

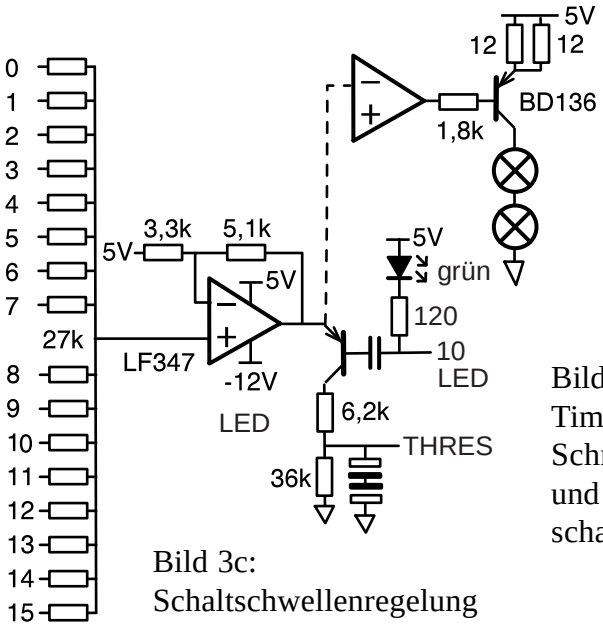


Bild 3c: Schaltschwellenregelung

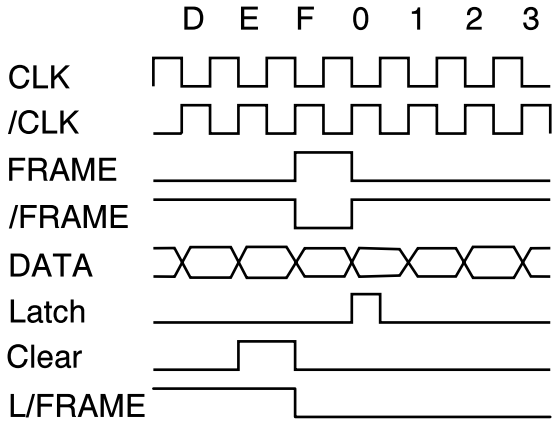
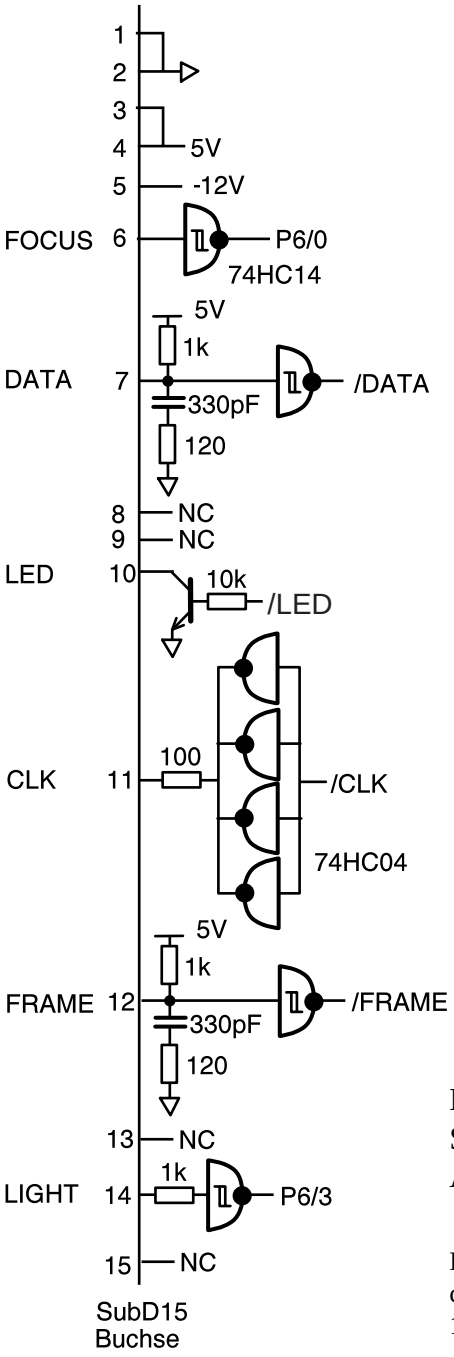


Bild 4: Timing Schnittstelle und Adapterschaltung



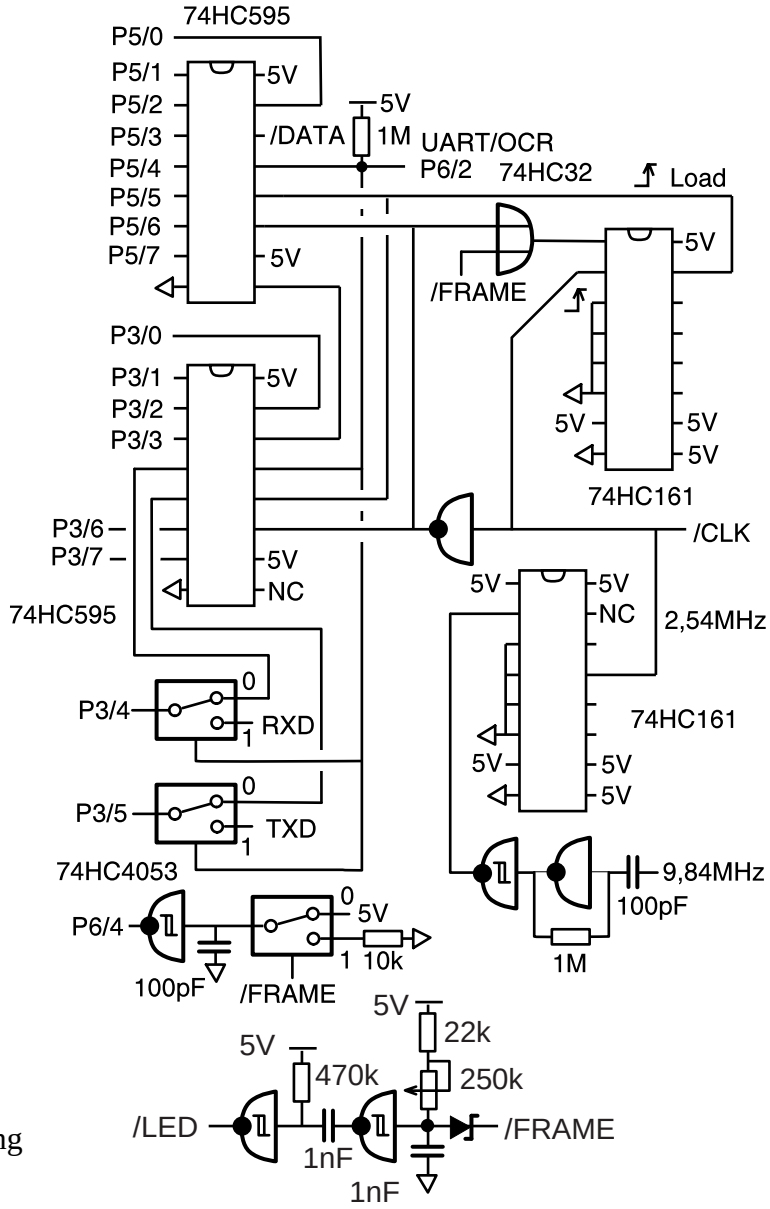


Bild 5: Schaltung Adapter

Der orginale Synchronisierungspuls FRAME ist zu kurz, als daß man ihn in Software an einem Port detektieren könnte. Er wurde deshalb in der Schaltung per Monoflop als L/FRAME auf 10usec verbreitet.

Software

Die Leseroutine (Listing CGK.F74) wartet auf den FRAME-Puls in einer Schleife:

```
1 $: 1 $ P6 4  BBs, \ 5cyc
```

Der sich ergebende Jitter von 0 - 5 cyc ist tolerierbar. Das Lesen eines 16 Pixel-Worts sieht dann so aus:

```
P3 LDA, hhhh     STA, \ 8cyc
P5 LDA, hhhh 1+ STA, \ 8cyc
```

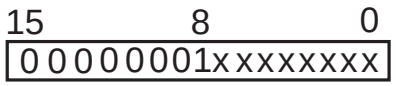
Nach 64d solcher Sequenzen ist ein Bild eingelesen. Das Teilprogramm ist damit etwa 700d Byte lang. Man beachte, daß bei den Pixel 0 = schwarz entspricht und einige Zeilen umgeordnet werden müssen.
Für provisorische Ausgabe als ASCII-Grafik aufs Terminal empfielt sich dieses auf VT52-Emulation einzustellen, damit man einen HOME-Befehl hat, der den Cursor nach rechts oben zurücksetzt. Da das Format eines Terminals 80 Zeichen x 25 Zeilen ist, ist es sinnvoll das 64x16 Pixel Bild liegend auszugeben. Bild 6 zeigt eine Ausdrucke, hier wieder stehend. Die Buchstaben müssen feste Grösse haben, etwa 3 x 1,5mm um voll ins Sichtfeld zu passen. Es hat sich außerdem ergeben, das zwar Buchstaben aus Büchern oder mit Bleistift geschriebene Zeichen guten Kontrast ergeben. Nicht jedoch schwarze Tinte oder Ausdrucke von Tintenstrahldrucker. Vermutlich verträgt die CCD den unterschiedlichen IR-Anteil nicht.

[1] emb 1 S. 15

Einfacher Logarithmus

Speziell für Dynamikkompressoren sind oft nur sehr grobe Näherungen nötig. Da die Datenworte dabei sehr breit sind, empfehlen sich Tabellen wegen des Speicherverbrauchs nicht.

Bild1: scanbit



scanbit

Die simpelste Näherung ist sich bei einer positiven Zahl das höchste gesetzte Bit zu suchen (Bild 1). Die Funktion gibt es bei manchen 32 Bit RISC-Prozessoren im Befehlssatz, weil für Normalisierung von Floats nützlich. Dafür wird manchmal der Name „scanbit“ verwendet. Im ARM heißt er `CLZ`, „Count leading Zeros“.

Entsprechend gibt es auch den Befehl „spanbit“ der die höchste gesetzte Null findet. Für negative 2er-Komplementzahlen. Allerdings ist in Software wohl eine bitweise Invertierung des Datenworts gefolgt von scanbit angemessener.

Die Werte von scanbit entsprechen der Funktion y=lb(x)+1. Wobei lb der binäre Logarithmus ist. Da Taschenrechner den manchmal nicht haben, ist die Umrechnungsformel über ln für Nachrechnen von Hand nützlich (Bild 2).

Etwas problematisch ist der hohe Datenverlust, wenn man 16 Bit auf 4 Bit staucht. (Listing SCNBIT.F74)

Tabelle 1:
y = scanbit(x)

x	y
0	0
1	0
2	1
4	2
8	3
16	4
32	5
...	
16384	15

Bild 3: bitlog 16

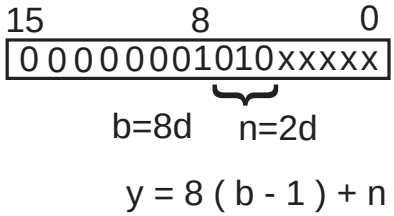


Bild 2: Umrechnung ln auf lb

lb(x) = 1,44269041 * ln (x)

bitlog

Man kann das Verfahren natürlich verfeinern indem man mehr Bits auswertet. Unter dem Namen bitlog ist eine solche Variante in [1] für 16 Bit Integer angegeben (Bild 3). Dabei werden die 3 folgenden Bits hinter dem führenden Bit zur Veredelung verwendet. Die Berechnung entspricht relativ gut y=8(lb(x)-1). Bei Eingangswerten 0 ... 7 treten allerdings Probleme auf sodaß ein Patch nötig ist (Tabelle 2) . Man beachte auch den Unterschied der Kennlinie zu echtem Logarithmus in diesem Bereich (Bild 4): letzterer läuft auf 1.

Die 16 Bit werden auf etwa 7 Bit gestaucht, der Maximalwert für y ist 119d. Das Verfahren läßt sich unschwer auf 32 und 64 Bit Datenworte skalieren (Tabelle 3). In Tabelle 4 sind Speicher und Laufzeit für Assemblerroutinen auf Mitsubishi-6502 mit 2,54 MHz Busfrequenz angegeben (Listings BL16.F74 , BL32.F74 BL64.F74)

Tabelle 2: y = bl16(x)

x	x binär	b	n	y
8	...1000	3	0	16
7	...0111			14
6	...0110			12
5	...0101			10
4	...0100			8
3	...0011			6
2	...0010			4
1	...0001			2
0	...0000			0

Expander

Aus beiden Verfahren lassen sich auch Umkehrfunktionen bilden die sich dann ähnlich Exponentilalfunktionen verhalten. Für diese sind wegen der kurzen Wortlänge von x aber oft Tabellen möglich, wodurch man beliebige Kennlinien erzeugen kann.

[1] Crenshaw „MATH Toolkit for REAL-TIME Programming“ CMP-Books 2000

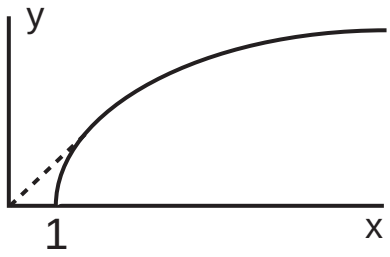


Bild 4: Kennlinien log & bitlog

Tabelle 4: bitlog

Speicher & Laufzeit	Byte	usec
BL16	60	25 - 100
BL32	90	50 - 150
BL64	120	50 - 250

Tabelle 3: Skalierung von bitlog von 16 auf 32 und 64 Bit

Y= 8(b-1)+n	b = 0 ...15d	n = 3 bit	X<8	Y= X*2	Ymax =	119d
Y=16(b-2)+n	b = 0 ...31d	n = 4 bit	X<16d	Y= X*2	Ymax =	479d
Y=32(b-3)+n	b = 0 ...63d	n = 5 bit	X<32d	Y= X*2	Ymax =	1951d

Compact Flash an Controller

Der ideale Weg um einen Einplatinencomputer mit so etwas wie einer Floppy auszustatten. D.h. einen wechselbaren Speicher mit dem man Files von und zu PCs übertragen kann.

Die Vorzüge gegenüber einer Floppy sind erheblich. Die Schnittstelle paßt direkt an den Bus einer 8 Bit CPU. Die Bauform ist klein. Es ist sowohl 3,3V als auch 5V Versorgung möglich. Stromaufnahme ist mit ca. 50mA erträglich.

Die beiden Varianten CF I und CF II unterscheiden sich nur in der Bauhöhe. Die dünnere CF I paßt auch in CF II Stecker. 1994 mit 4 MByte Karten eingeführt denen 1996 die heute übliche kleinste Grösse 8 MByte folgte. Übliche Speicherdichten für kleine Varianten sind 8, 16, 32 MByte. Verfügbar, aber derzeit noch teuer, sind auch wesentlich größere Audführungen. Da ursprünglich für Laptops gedacht paßt der 50pol Steckverbinder über einen passiven Adapter in PCMCIA-Karten. Neben dieser Anschaltung ist alternativ „True IDE“ möglich. In diesem Fall emuliert man genauso direkt eine Harddisk. Hier ist aber „hot plugging“, also Stecken und Entfernen der Karte bei laufendem Gerät nicht vorgesehen. Auf diese wünschenswerte Eigenschaft will man aber ungern verzichten. Die dritte Beschaltung „Common Memory“ ist für Controller besonders geeignet. Man hängt direkt memory mapped am Bus der CPU.

[1] www.compactflash.org
„CF+ and CompactFlash Specification Revision 2.0“
[2] www.sandisk.com
„Compact Flash Memory Card Product Manual“
„Using SanDisk Flash ATA Components with an 8051 Microcontroller“
[3] www.t13.org
„Working Draft X3T10/0948D“ ATA-2
[4] www.renesas.com
„Hitachi Flash Cards User’s Manual“
„Q&A on Hitachi CF and ATA Cards“

Hardware

Wegen des benötigten RAMs ist ein Einplatinencomputer sinnvoller als ein Einchip-Controller. Die CF-Karte liegt dann wie ein Peripheriebaustein im Speicher. Wegen der Zugriffszeit von 150nsec darf der Bus nicht zu schnell sein.

Damit „hot plugging“ möglich ist, muß er über Treiber die man tristate schalten kann abgetrennt sein (Bild 1). Das verbessert auch den ESD-Schutz etwas. Die Treiber sind zudem nötig, weil an der Schnittstelle nicht TTL sondern CMOS-Pegel gefordert sind. Die Beschaltung decodiert 16 Register, benötigt werden typisch aber nur 8. Also könnte man A3 auch auf GND legen. Neben der Busanschaltung sind auch noch einige Portpins erforderlich.

Mit `/EN` schaltet die CPU die Versorgung des Speichers an.

Über den Pin `RESET` kann er den Speicher rücksetzen. Die Pulsbreite muß größer 10usec sein, es kann im Betrieb 100msec dauern bis die Karte wieder bereit ist. Unmittelbar nach PowerUp kann es auch 400msec dauern. Typische Zeiten schwanken von 5 - 250msec.

Zwar haben die Karten nominell eine interne Resetschaltung, aber normalerweise wird man bei Anlegen der Versorgung `RESET=1` schalten und erst nach Verzögerung mit `RESET=0` freigeben. Wenn man den Pin nicht ansteuern will verbindet man ihn fest mit GND.

Der `RDY/BSY`-Pin zeigt an wann die Karte den Reset beendet hat und bereit ist. Der Pin muß nicht zwingend auf Port gelegt sein, man kann das Bit auch in einem Register abfragen.

`/CD1` ist im Speicher mit GND verbunden. Damit prüft die CPU ob eine Karte steckt. Sie wird dann die Versorgung anschalten und damit auch die Treiber freigeben.

Schaltung

Der hier verwendete Mitsubishi-6502 führt jeden Schreibzugriff als read-modify-write Zyklus aus. Das macht bei Peripheriebausteinen fast immer Probleme. Deshalb muß man Schreiben und Lesen in unterschiedliche Speicherbereiche legen und die Decodierung entsprechend ausführen (Bild 2).

Die beiden Ports sind hier 74HC-Bausteine. Beim Register 74HC175 wird durch den Reset `/RES` der CPU die Schnittstelle inaktiv geschaltet. Da die Stromaufnahme der Karte typisch nur etwa 50mA beträgt genügt ein Wald&Wiesen-Transistor als Schalter.

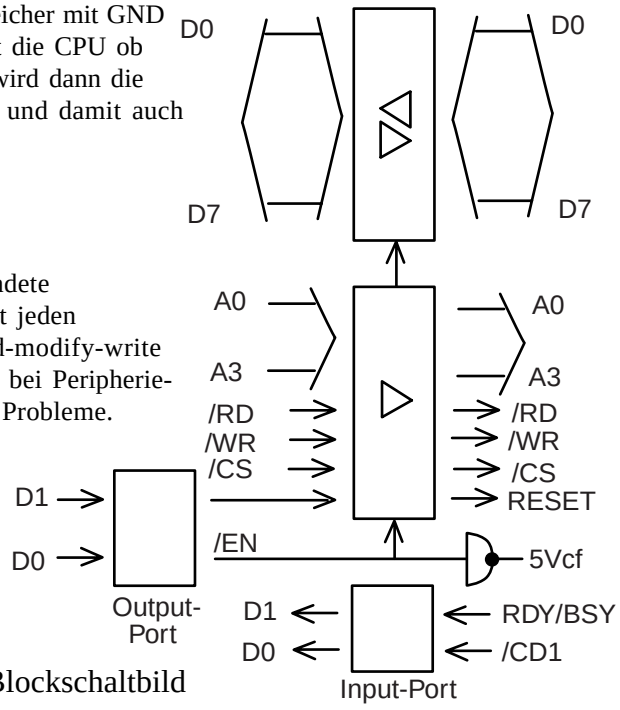


Bild1: Blockschaltbild

Tabelle 1: Pins der CF-Karte in Memory Mode sowie deren Beschaltung

1	GND		GND	26	/CD1	O		/CD1
2	D3	I/O	D3cf	27	D11	I/O		nc
3	D4	I/O	D4cf	28	D12	I/O		nc
4	D5	I/O	D5cf	29	D13	I/O		nc
5	D6	I/O	D6cf	30	D14	I/O		nc
6	D7	I/O	D7cf	31	D15	I/O		nc
7	/CE1	I	r /CScf	32	/CE2	I	r	5Vcf
8	A10	I	GND	33	/VS1	O		nc
9	/OE	I	r /RDcf	34	/IORD	I	r	nc
10	A9	I	GND	35	/IOWR	I	r	nc
11	A8	I	GND	36	/WE	I	r	/WRcf
12	A7	I	GND	37	RDY/BSY	O		RDY/BSY
13	Vcc		5Vcf	38	Vcc			5Vcf
14	A6	I	GND	39	/CSEL	I		GND
15	A5	I	GND	40	/VS2	O		nc
16	A4	I	GND	41	RESET	I	r	RESETcf
17	A3	I	A3cf	42	/WAIT	O		nc
18	A2	I	A2cf	43	/INPACK	O		nc
19	A1	I	A1cf	44	/REG	I	r	5Vcf
20	A0	I	A0cf	45	BVD2	I/O		nc
21	D0	I/O	D0cf	46	BVD1	I/O		nc
22	D1	I/O	D1cf	47	D8	I/O		nc
23	D2	I/O	D2cf	48	D9	I/O		nc
24	WP	O	nc	49	D10	I/O		nc
25	/CD2	O	nc	50	GND			GND

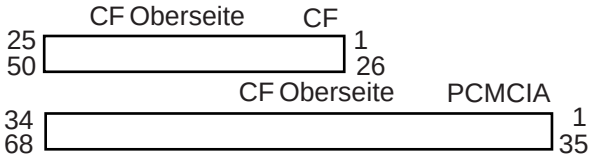
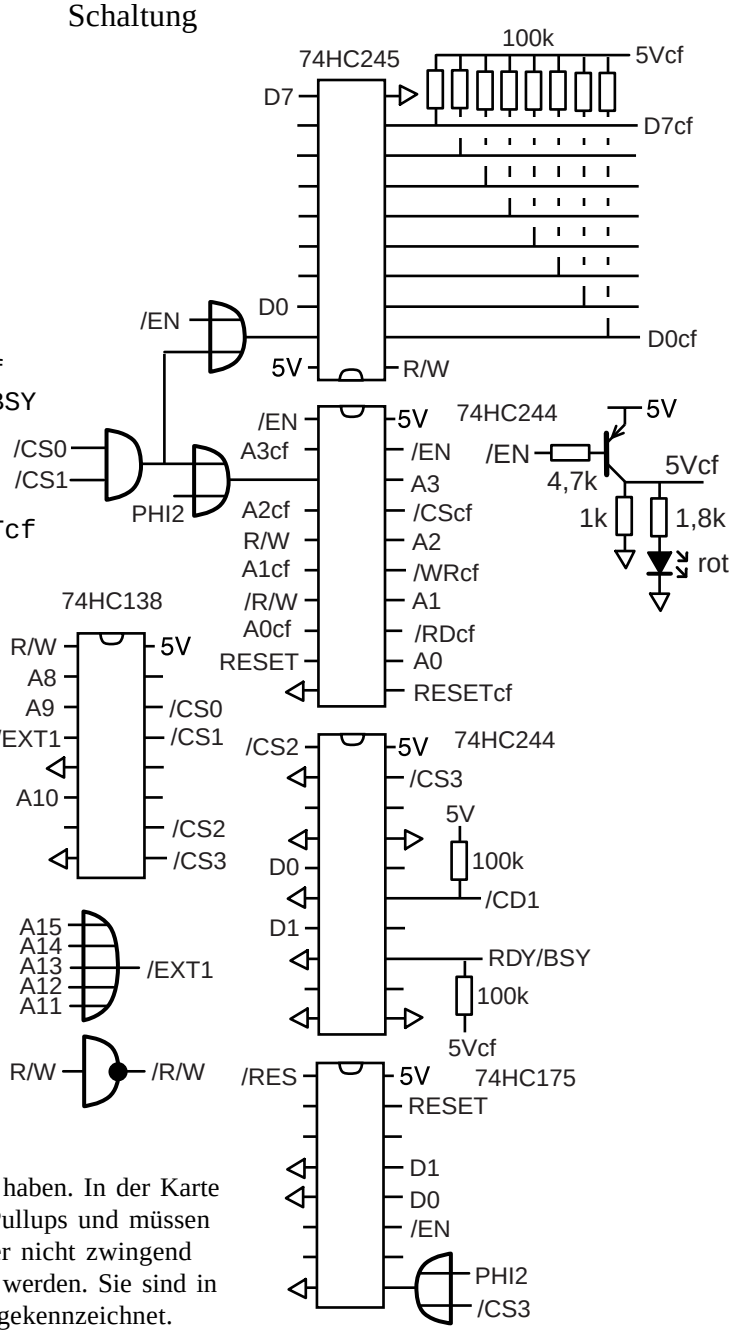


Bild 3: Nummerierung der Pins
Blick auf Rückseite der Stecker

Bild 2: Schaltung



Echte Harddisks in CF-Bauform könnten jedoch bis 500mA benötigen und werden hier nicht berücksichtigt. Das rote LED zeigt dem Benutzer an das die Karte unter Spannung steht und nicht gezogen werden darf.

Die Pullups an 74HC244 und 74HC245 werden gebraucht, damit sie bei gezogener Karte definierte Pegel am Eingang haben. In der Karte haben viele Pins Pullups und müssen deshalb am Stecker nicht zwingend mit 5V verbunden werden. Sie sind in Tabelle 1 mit „r“ gekennzeichnet. Dort ist links der offizielle Name sowie Signaltyp und rechts das in der Beschaltung angelegte Signal vermerkt.

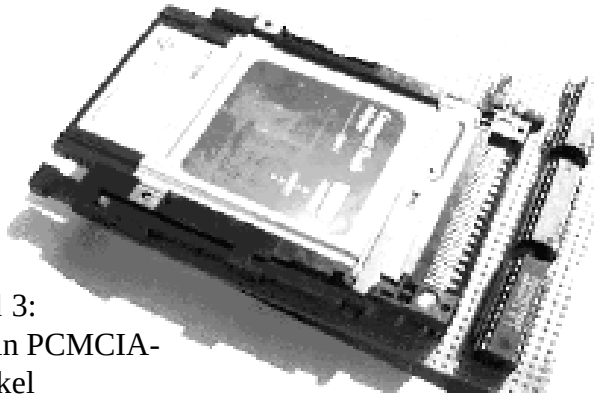


Bild 3:
CF in PCMCIA-Sockel

Sockel

Die üblichen CF-Sockel mit ihren 0,64mm SMD-Kontakten sind für Breadboards ungünstig. Von den PCMCIA-Sockeln gibt es eine Variante die in Laptops huckepack über dem unteren Sockel auf der Leiterplatte montiert wird (Bild 3). Dessen bedrahteten Pins haben dann 2,54mm Raster. Es wird allerdings zusätzlich ein handelsüblicher PCMCIA auf CF Adapter benötigt. Dessen modifizierte Pinbelegung ist im Anhang der CF-Spec angegeben [1] (Tabelle 2).

Tabelle 2:
Adapter PCMCIA auf CF

PCM CIA	CF				
1	1	GND	35	1	GND
2	2	D3	36	26	/CD1
3	3	D4	37	27	D11
4	4	D5	38	28	D12
5	5	D6	39	29	D13
6	6	D7	40	30	D14
7	7	/CE1	41	31	D15
8	8	A10	42	32	/CE2
9	9	/OE	43	33	/VS1
10	-		44	34	/IORD
11	10	A9	45	35	/IOWR
12	11	A8	46	-	
13	-		47	-	
14	-		48	-	
15	36	/WE	49	-	
16	37	RDY/BSY	50	-	
17	13	Vcc	51	38	Vcc
18	-		52	-	
19	-		53	-	
20	-		54	-	
21	-		55	-	
22	12	A7	56	39	/CSEL
23	14	A6	57	40	/VS2
24	15	A5	58	41	RESET
25	16	A4	59	42	/WAIT
26	17	A3	60	43	/INPK
27	18	A2	61	44	/REG
28	19	A1	62	45	BVD2
29	20	A0	63	46	BVD1
30	21	D0	64	47	D8
31	22	D1	65	48	D9
32	23	D2	66	49	D10
33	24	WP	67	25	/CD2
34	50	GND	68	50	GND

Zugriff

Spätestens in Software hat man das modifizierte ATA-Interface vor sich. Studium von ATA-2 [3] als Vergleich ist empfehlenswert. Die „Common Memory“-Betriebsart ist letztlich eine Untermenge davon. Es sind zwar mehr Register vorhanden, praktisch genügen aber die ersten 8 Register des „task files“ (Tabelle 3).

Der eigentliche Speicher ist fest in Sektoren zu 512 Byte eingeteilt. Diese Daten kommen sequentiell per Protokoll durch das DATA-Register. Im echten ATA-Interface ist der Datenbus 16 Bit breit. Bei dem hier vorliegenden 8 Bit Interface

Tabelle 3: Register „task-file“

Adr	Read	Write
x0	DATA	
x1	ERROR	FEATURES
x2	SECTOR-COUNT	
x3	SECTOR-NUMBER	
x4	CYLINDER-LO	
x5	CYLINDER-HI	
x6	DRIVE/HEAD	
x7	STATUS	COMMAND

kommen die Datenworte little-endian, was für den 6502 gut paßt, aber bei big-endian Controllern zu berücksichtigen ist.

Wenn man beim Zugriff nicht nur einen, sondern mehrere Sektoren übertragen will stellt man das vorher in SECTOR-COUNT ein:

00h 256 Sektoren
01h 1 Sektor
02h 2 Sektoren
...

Hier wird immer nur 1 Sektor übertragen, also Einstellung 01h.

In den Registern x3 bis x6 wird die Adresse des ersten Sektors vorgegeben. Es gibt dafür zwei Varianten: CHS („Cylinder, Head, Sector“) und LBA („Linear Block Adressing“). Hier wird die einfachere LBA-Adreßierung verwendet. Sie ist simpel ein 28 Bit Adreßwort das so über die Register verteilt ist:

	bit
SECTOR-NUMBER =	7 - 0
CYLINDER-LO =	15 - 8
CYLINDER-HI =	23 - 16
DRIVE/HEAD =	27 - 24

Die obersten 4 Bit von DRIVE/HEAD müssen fest auf 1110b eingestellt werden.

Einschalten der Versorgungsspannung erfolgt hier (Listing CF1.F74) mit UP , Abschalten mit DOWN . Mit dem Befehl TF. kann man die Register auslesen. Das Ergebnis sieht ca. so aus:

```
 0  1  2  3  4  5  6  7
00 01 01 01 00 00 E0 50
```

Wichtig ist der Wert 50h im STATUS-Register, er zeigt an das die Karte bereit für Befehle ist (Tabelle 4). Das ERROR-Register kann man als Erweiterung von STATUS ansehen. Im normalen Betrieb ist für Zugriff das BUSY-Flag wichtig. Sowie das Flag DRQ das anzeigt wann ein Datenbyte gelesen bzw. geschrieben werden kann.

Wenn Zugriff möglich ist, lädt man in Register x1 - x6 die passenden Einstellung. Soweit die LBA-Adresse nicht komplett erforderlich ist, muß man nach Reset zumindest die oberen 4 Bit von DRIVE/HEAD initialisieren. Dann schreibt man den Befehls-Token in das COMMAND-Register wodurch der Befehl ausgeführt wird.

Von den 40 Befehlen des Standards sind normalerweise in der CF-Karte nur 30 implementiert. In der Ansteuerungssoftware hier auf 4 Befehle reduziert. Die Übergabe der LBA-Adressen erfolgt dabei nicht über Stack sondern die 32 Bit Variable LBA . Wenn man sich auf 32 MB Karten beschränkt sind nur die unteren 16 Bit nötig und alles oberhalb statisch 000h.

Tabelle 4: STATUS-Register, ERROR-Register

	Bit	
STATUS	7	1 = BUSY
	6	1 = RDY Ready (wait for RDY=1 after Reset)
	5	1 = DWF Write Fault
	4	1 = DSC Ready
	3	1 = DRQ Data Request
	2	1 = CORR Correctable Error was corrected
	0	1 = ERR Error in Register ERROR
ERROR	7	1 = Bad Block
	6	1 = Uncorrectable Error
	4	1 = ID error
	2	1 = Abort , invalid command
	0	1 = General Error

Die Variable >CACHE zeigt auf den Beginn das Speichers im RAM des Controllers von dem der Sektor gelesen, bzw. wohin der Sektor geschrieben wird. Ihre unteres Byte ist immer 00h.

Daten lesen

```
READ-SECTOR  \ (  –  )
             \ opcode 20
```

Die LBA-Variablen entsprechend der Adresse des ersten Sektors werden in die Register x3 bis x6 der Karte kopiert. In SECTOR-COUNT speichert man die Zahl der Sektoren die man lesen will. Hier soll immer nur ein Sektor übertragen werden, also 01h. Dann schreibt man den Befehlscode 20h und wartet darauf, das in STATUS das BSY-Flag gelöscht und das DRQ-Flag gesetzt ist. Während des Lesens der folgenden 512 Byte aus DATA müssen die Flags in STATUS nicht geprüft werden.

Daten schreiben

```
WRITE-SECTORS  \ (  –  )
               \ opcode 30
```

WRITE-SECTORS funktioniert wie READ-SECTORS mit geänderter Richtung der Datenbewegung. Das Schreiben der Bytes einzelner Sektoren erfolgt ungebremst, weil diese im RAM der Karte gepuffert werden. Da das Schreiben von Flash aber dauert, kann dann eine Pause von 700usec („Class 2 Befehl“) auftreten bis das DRQ-Flag wieder gesetzt wird. Hier wird explizit gewartet bis die Karte wieder bereit ist. Sonst könnt man z.B. mit DOWN irrtümlich die Versorgung abschalten während die Karte noch am speichern ist.

Einstellungen lesen

```
IDENTIFY-DRIVE \ (  –  )
               \ opcode EC
```

Ähnlich wie mit dem Read-Befehl werden 512d Byte Einstellungen und Herstellerinformationen aus der Karte ausgelesen (Tabelle 5). Speziell für Formatierung kann man so die Speichergrösse bestimmen.

```
IDENTIFY-DRIVE. \ (  –  )
```

ist der zugehörige Druckbefehl (Tabelle 6) am Beispiel einer 8 MB Karte.

Tabelle 5: Identify Drive

Wrd Adr	Default Data	Bytes	
0	848Ah	2	general configuration bits
1	xxxx	2	default number of cylinders
2	0000	2	reserved
3	xxxx	2	default number of heads
4	0000	2	unformatted bytes per track (obsolete)
5	0240	2	unformated bytes per sector (obsolete)
6	xxxx	2	default number of sectors per track
7	xxxx	4	MSW number of sectors per card
8			LSW
9	0000	2	reserved
10	aaaa	20d	serial number in ASCII right justified
20	0002	2	buffer type = dual ported (obsolete)
21	0002	2	buffer size in 512 byte increments: 1k (obsolete)
22	0004	2	# of ECC bytes passed on R/W long
23	aaaa	8	firmware revision (big endian)
27	aaaa	40d	model number (big endian)
47	0001	2	maximum of 1 sector on R/W multiple
48	0000	2	double word not supported (obsolete)
49	0200	2	DMA not supported , LBA not supported
50	0000	2	reserved
51	0200	2	PIO data transfer cycle timing mode
52	0000	2	DMA data transfer cycle tim. not supported (obsolete)
53	0003	2	field validity
54	0003	2	current number of cylinders
55	xxxx	2	current number of heads
56	xxxx	2	current sectors per track
57	xxxx	4	LSW current capacity in sctors
58			MSW
59	010xh	2	multiple sector setting is valid
60	xxxx	4	total number of sectors addressable
61			in LBAmode
62	xxxx	4	reserved
63			
64	0003	2	advanced PIO modes supported
65	0000	4	reserved
66			
67	0078	2	minimum PIO transfer without flow control
68	0078	2	mimimum PIO transfer with IORDY flow control
69	0000	130d	reserved
128	0000	64d	reserved vendor unique bytes
160	0000	192d	reserved

Standby

```
SLEEP   \ (  –  ) opcode 99
```

Reduziert den Stromverbrauch auf <1mA. Reaktivierung erfolgt durch beliebigen Befehl. Da echte Harddisks Stromfresser sind hatte ATA diverse Befehle für Standby definiert. Weil

sich CF-Karten automatisch recht schnell runterschalten sind solche Befehle hier fast nicht nötig. Abschalten der Versorgung ist wegen der langsamen Einschaltzeit vieler Karten aber auch unattraktiv.

Einschränkungen

Die Schaltung funktioniert nicht für alle Karten. Ein Exemplar Casio 8MB mit Hitachi-Innereien war nicht ansprechbar. Funktioniert aber an Kartenleser am PC. Erklärung findet sich eventuell in Angaben im Hitachi-Handbuch zur Behandlung des /OE-Pins beim Powerup. Der Aufwand auch pathologische ältere Karten zu berücksichtigen scheint aber nicht gerechtfertigt.

(Fortsetzung von S.20 „Lineare Interpolation“)

Diese Umrechnung kann man durch eine passende Routine automatisch vornehmen: maximalen Fehler an der Stützstelle und den 3 davorliegenden interpolierten Punkten berechnen und abspeichern. Dann den Wert an der Stützstelle dekrementieren bzw inkrementieren (Je nach Vorzeichen der Abweichung an den Stützstellen). Darauf nochmal den maximalen Fehler an allen 4 Punkten berechnen. Ist er größer geworden als der gespeicherte Wert den letzten Schritt rückgängig machen und Optimierung beenden. Als alternative Abbruchbedingung wird geprüft ob das Vorzeichen des Fehlers wechselt. D.h. ob man den Wert an der Stützstelle so weit verschoben hat, daß er selbst nun den maximalen Fehler enthält. Was natürlich nicht sein soll (Listing MLINT.F74).

Man sollte sich anhand der 4kByte Tabelle als Referenz den Fehler der berechneten Punkten ausdrucken (Tabelle 1). Bei der Funktion x^2,5 ist der relative Fehler anfangs extrem hoch. Am Ende der Tabelle wird er geringer. In der beabsichtigten Anwendung waren Werte unterhalb 50h und oberhalb 200h von geringer Bedeutung. Sodaß keine weiteren Optimierungen nötig waren.

Tabelle 6: Test IDENTIFY-DRIVE

```
— — — | UP                \ switch power on
— — — | IDENTIFY-DRIVE    \ read data
— — — | IDENTNTIFY-DRIVE.  \ print data

general configuration bits                          848A
default number of cylinders                         00F6
default number of heads                             0002
number of unformatted bytes per track               0000
number of unformated bytes per sector               0200
default number of sectors per track                 0020
number of sectors per card                          0000 3D80
serial number                                       X0102
20030724025939
buffer type = dual ported                 = 0002    0002
buffer size in 512 byte increments                  0002
# of ECC bytes passed on R/W long                   0004
firmware revision                                   Rev 2.00
model number
                                                 Hitachi XXM2.2.0
maximum of 1 sector on R/W multiple                 0001
double word not supported                  = 0000   0000
DMA not supported , LBA not supported      = 0200   0200
PIO data transfer cycle timing mode        = 0200   0100
DMA data transfer cycle tim. not supported   =0      0000
field validity                                      0001
current number of cylinders                         00F6
current number of heads                             0002
current sectors per track                           0020
current capacity in sectors                         0000 3D80
multiple sector setting is valid           = 010x   0100
total number of sectors addressable in LBAmode      0000 3D80
advanced PIO modes supported               = 0003   0000
minimum PIO transfer without flow control= 0078     0000
mimimum PIO transfer with IORDY flow cntrl 0078     0000
— — — | DOWN  \ power down
```

(Fortsetzung von S.17 „Fletcher Prüfsumme“)

Adler-32

In RFC-1950 gibt es von 16 Bit Fletcher eine aufwendige Variante die bei etwas geringerer Fehlersicherheit immer noch effizienter als CRC32 in Software sein soll.

Bei Fletcher wird der Modulo-überlauf der beiden Akkumulatoren ignoriert. Man kann den Inhalt der Akkumulatoren auch als Rest einer Division durch 65536 interpretieren. Bei Adler ist der Rest einer Division durch die Primzahl 65521 vorgesehen, In Bild 3 durch den Befehl „mod“ der nur den Rest als Ergebnis hat dargestellt. Durch passende Auslegung des Programms ist es wohl möglich die Division nur nach 5552 Bytes ausführen zu müssen. Trotzdem scheint das Verfahren für kleine Controller nicht attraktiv.

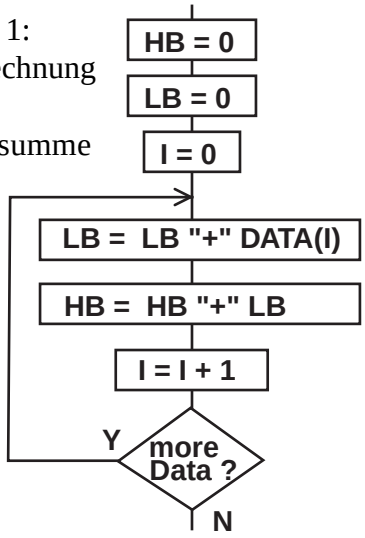
[1] Fletcher „An Arithmetic Checksum for Serial Transmission“ IEEE Trans. on Communications Jan. 1982

[2] Sklower „Improving the Efficiency of the OSI Checksum Calculation“ ACM Computer Communication Review Okt. 1989

Fletcher Prüfsumme

In Software ohne Tabellen schneller berechenbar und fast so wirksam wie CRC.

Bild 1: Berechnung der Prüfsumme



Erste breitere Anwendung erfolgte in OSI-Protokollen. Da Pakete aus Bytes bestehen, wurde hier als Wortbreite der Daten 8 Bit verwendet. Wie auch in [1] als gängigste Variante angenommen. Bild 1 zeigt schematisch den Ablauf. Es gibt zwei Akkumulatoren LB und HB deren Länge dem Datenwort, also einem Byte entspricht. Nach Ende der Berechnung werden die Akkumulatoren zur Prüfsumme, hier also mit 16 Bit Länge zusammengesetzt. Typisch wird diese little-endian dem Paket angehängt.

Es sind nur Additionen erforderlich, aber Berechnung in 1er-Komplement ist besser als in 2er-Komplement und deshalb üblich [1]. In Bild 1 ist diese etwas andere Addition durch „+“ gekennzeichnet.

1er-Komplement

Eine recht ungebräuchliche Variante der Zahlendarstellung (Bild 2), die Arithmetik üblicher CPUs verarbeitet 2er-Komplement direkt. Für 1er-Komplement ist hier die nachträgliche Addition des Carrybits nötig. Beispiel:

```
  FE  \   -1
+ 04  \ + +4
-----
0102  \ 02 ; carry

  02
+ 01  \ add carry
-----
  03  \ = +3
```

Bei 8 Bit CPUs damit nur ein zusätzlicher Opcode fällig:

```
FE #. LDA, \ -1
04 #. ADD, \ +4
00 #. ADC, \ +carry
```

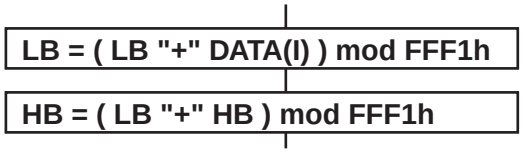
Da man in FORTH mit 16 Bit Wortbreite rechnet, kann man sich kompliziertere Programme ausdenken, bei denen das Carrybit von bis zu 255d Bytes akkumuliert wird, bevor man es in einem Korrekturschritt zum unteren Byte summiert. Auf einem Controller dürfte jedoch der simple Algorithmus in Assembler eine deutlich effizientere Löung sein.

Null

Wenn man bei CRC mit Startwert Null alle Bytes eines Pakets und die Prüfsumme addiert ist das korrekte Ergebnis Null. Es hat sich eingebürgert dieses Verhalten auch anderen Prüfsummen aufzuzwingen. Dazu muß man die Prüfsumme passend doktern, bevor man sie dem Paket anhängt. Als Grund dafür wird angegeben, daß Mehraufwand im Sender sinnvoll ist, wenn er zu geringerem Aufwand im Empfänger führt. Denn ein Paket wird nur einmal erstellt, aber beim Weg durch ein Datennetz wird die Prüfsumme oft überprüft.

Wenn alle Bytes des Pakets Null sind, ist auch die Prüfsumme

Bild 4: Adler-Variante



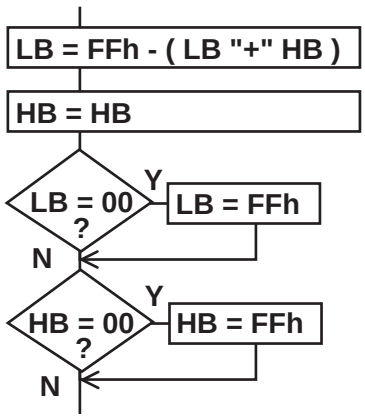


Bild 3: Modifikation des Endwerts

Null. Auch dagegen gibt es Einwände, sodaß für Fletcher ein Patch üblich ist 00 auf FF zu ändern. Beide Modifikationsschritte sind in Bild 3 dargestellt. Nur im Sender benötigt, Empfänger „addiert“ beide Bytes konventionell um das Ergebnis Null zu erhalten.

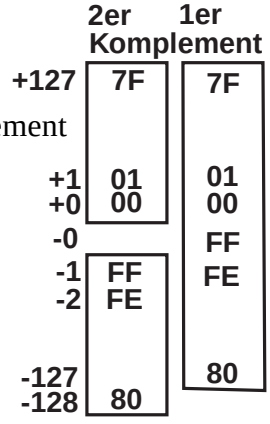
Im Listing FLET1.F74 in 6502-Assembler für 8 Bit Fletcher als Prüfsumme für Speicher aufgeführt.

Fletcher-16

Für Internetanwendungen z.B. RFC-1146 wird alternativ zu 8 Bit die 16 Bit Fletcher mit 32 Bit Prüfsumme favorisiert. Weil auf großen CPUs effizienter [2]. Die Akkumulatoren sind nun 16 Bit breit, am Rechenschema ändert sich nichts. Da aber nun 16 Bit Worte eingelesen werden, muß man in Paketen ungerader Länge das letzte Byte um 00h ergänzen. Big/little-endian-Probleme sind natürlich auch noch möglich. Anders als bei ISO wird die Prüfsumme nicht so modifiziert, daß die Summe Null ergibt.

(Fortsetzung auf S. 16)

Bild 2: 1er/2er Komplement 8 Bit



Hashing für RFIDs

RFIDs sind über Funk abfragbare Seriennummer-ICs. Hashing ist ein Verfahren für effizientes Suchen in Listen.

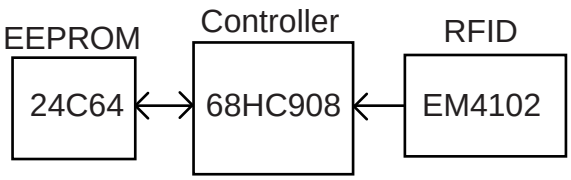
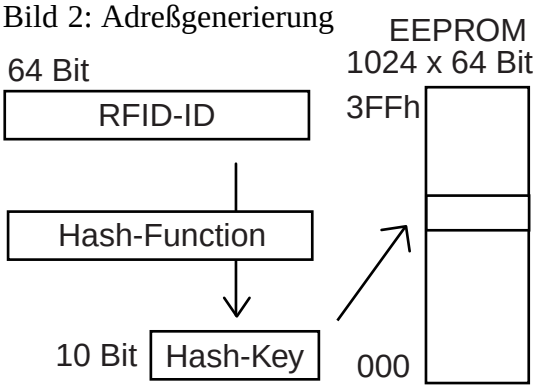


Bild 1: Lesegerät

Bild 2: Adreßgenerierung



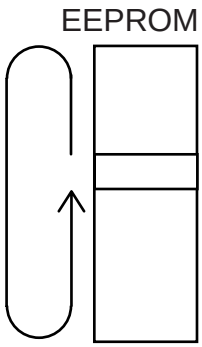


Bild 3: Suche

Die preiswertesten 125kHz Transponder (RFID) haben eine feste Nummer die vom Hersteller z.B. per Laser eingeschrieben wird. Damit dieser trotz kontinuierlicher Fertigung eine „einzigartige“ Seriennummer garantieren kann, ist diese ziemlich lang, typisch 64 Bit. Da darin auch Prüfsummen enthalten sind, ist die effektive Länge kürzer. Um Portierung zwischen verschiedenen Anbietern zu vereinfachen ist es für ein Lesegerät trotzdem günstiger mit der vollen Länge von 64 Bit zu arbeiten.

Bei Anwendung in größeren Gebäuden und Hotels kann es erforderlich sein, daß das Lesegerät hunderte von Schlüssel erkennen soll. Wenn diese in einem externen seriellen EEPROM gespeichert sind, ist der Zugriff langsam (Bild 1). Um die Reaktionszeit des Lesegeräts kurz zu halten muß die Suche optimiert werden.

Hashing

Das Suchen in ungeordneten langen Listen ist ein altes Problem der Datenverarbeitung und eine bekannte Lösung heißt Hashing. Durch eine geeignete Hash-Funktion wird aus dem 64 Bit Datensatz des Transponders ein 10 Bit Wert berechnet, der als Zeiger in die Liste mit den 1024 x 64 Bit Worten dient (Bild 2). Dort sucht man nun linear aufwärts weiter. Wenn oberhalb 3FF ein Moduloüberlauf eintritt, geht die Suche ab Adresse 000 weiter (Bild 3). Der Endpunkt ist der ursprüngliche Einsprungpunkt auf den der Hash-Key zeigt. Normalerweise muß man aber nur ein kurzes Stück linear suchen. Beim Einprogrammieren eines Transponders sucht man die nächste leere Speicherzelle, also alle 64 Bits gesetzt. Dort programmiert man die RFID-ID ein. Beim Suchen vergleicht man die RFID-ID des Transponders mit dem Inhalt der Speicherzelle auf Identität. Bzw. ob die Speicherzelle leer ist. In letzterem Fall ist die ID nicht im EEPROM.

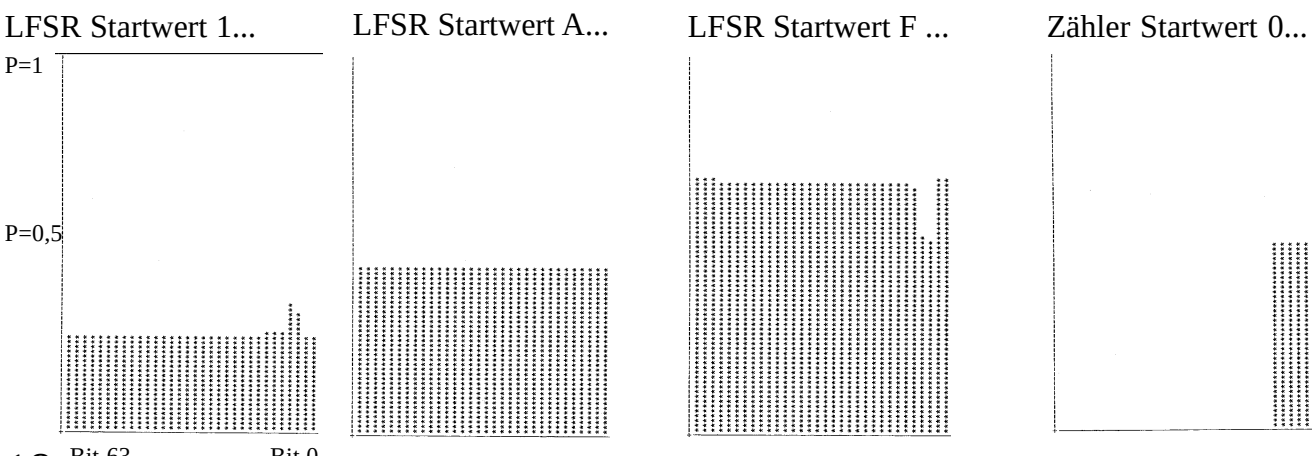
Das Verfahren reduziert die Suche in einer langen Liste auf die Suche in vielen kurzen Listen. Diese entstehen zwangsläufig, da Hash-Kollisionen, also mehrere IDs zeigen auf eine Teilliste, nicht vermeidbar sind.

Weiter entwickelte Hash-Verfahren führen statt linearer Suche noch einen 2. Hash-Schritt durch, aber das ist hier zu kompliziert. Besonders weil man aus dem seriellen EEPROM nach der Startadresse die folgenden Bytes kontinuierlich lesen kann, während man für Sprünge nochmal Kopf und Adresse schreiben muß.

Effizienz

Die Wirksamkeit des Verfahrens hängt von der Statistik der Eingangsdaten und der Anpassung der Hash-Funktion an diese ab. Ferner davon wie voll der Speicher ist.

Bild 4: ID-Generatoren



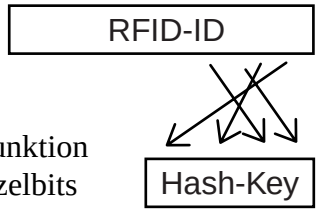


Bild 3: Hash-Funktion aus Einzelbits

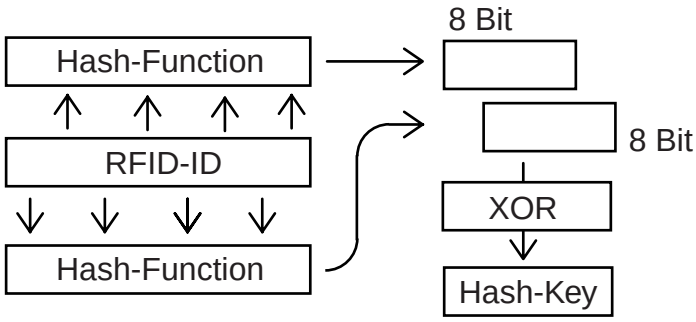


Bild 5: Verknüpfung von Hash-Funktionen durch XOR

Wegen des statistischen Einflusses kann man die Reaktionszeit des Systems nur per Monte Carlo Simulation verifizieren. Und hängt stark davon ab wie gut man die RFID-IDs nachbilden kann.

ID-Statistik

Für die Produktion von IDs gibt es zwei Varianten: simple Binärzähler oder Pseudozufallszahlen. Wenn letztere z.B. durch LFSR mit m-Sequenz erzeugt werden, ergeben sich auch einzigartige IDs. Die Hersteller von Transpondern machen keine Angaben wie sie ihre IDs erzeugen. Es scheint sich aber um Zufallszahlen zu handeln. Von Microchip ist für PICs bekannt, daß sie LFSR mit wenig Taps verwenden [1]. Insofern liegt man mit dem simplen 64 Bit Polynom von Stahnke [1] für Simulation wohl durchaus richtig.

Ein simpler Verteilungstest ist für alle 64 Bits die Wahrscheinlichkeit zu ermitteln für die das Bit den Wert 1 hat. Dazu lässt man den Generator z.B. 1024 Samples erzeugen und zählt dabei in 64 Zählern mit ob das jeweilige Bit 1 war. Für einen echten, guten Zufallszahlengenerator wäre der Wert überall 50%. Beim 64 Bit LFSR a la Stahnke ist die Verteilung etwas vom Startwert abhängig (Bild 4). Das liegt auch daran, daß die Stichprobe zu klein ist: nur 1024 von (2^64)-1 Worten. Beim Binärzähler ist die Schlagseite offensichtlich. Nur die untersten Bits ändern sich.

Hash-Funktionen

Simpelste Variante ist einfach nur bestimmte Bits zu kopieren (Bild 3). Geeignet sind Bits die möglichst mit 50% Wahrscheinlichkeit den Wert 1 haben. Beim Binärzähler sind das die untersten Bits. In der Hash-Adresse haben die oberen Bits die größte Wirkung, also wird man sie dorthin legen. Entsprechnd würde man höherwertige Bits der ID geringer bewerten. Das Verfahren funktioniert also nur sicher, wenn die Statistik des Eingangssignals bekannt und verläßlich ist.

In der Praxis ist das selten der Fall, robustere Verfahren sind nötig. Z.B. solche zur Prüfsummenberechnung. In [2] schneiden CRCs , Flechter und simples XOR über alle Bytes recht gut ab. Bezüglich Fehlersicherung ist die stark abfallende Qualität dieser Verfahren bekannt. Für Hashing sind die Qualitätsunterschiede aber nicht so deutlich. Ein Problem bei der Implementierung ist, daß für die Hash-Adresse der krumme

Tabelle 1: Ergebnisse Simulation

```
a = LFSR Startwert: 1000 0000 0000 0000 0000 0000 0000 0000
b = LFSR            FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
c = LFSR            AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
d = Zähler          0000 0000 0000 0000 0000 0000 0000 0000

CRC
  0    1    2    3    4    5    6    7    8    9    A    B    C    D    SUM
a 0243 0088 009D 004D 002A 0018 0008 0001 0000 0000 0000 0000 0000 0000 033F
b 0245 0089 0091 005B 0026 0016 0008 0002 0000 0000 0000 0000 0000 0000 034F
c 0246 0086 008F 0060 0027 0017 0004 0003 0000 0000 0000 0000 0000 0000 0343
d 0001 03FE 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001

FLETCHER
a 0183 0162 00C8 0042 000D 0004 0000 0000 0000 0000 0000 0000 0000 0000 01A0
b 0172 0189 00B3 003D 0011 0002 0002 0000 0000 0000 0000 0000 0000 0000 01A1
c 0171 0178 00CF 003A 000A 0004 0000 0000 0000 0000 0000 0000 0000 0000 018B
d 00C8 027B 00B2 000B 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 00C8

XOR
a 0286 0097 0067 002B 0022 0005 000F 0009 0006 0003 0001 0002 0000 0000 0B7D
b 0312 0026 0033 002A 0018 0019 0011 000D 0007 0008 0002 0001 0004 0002 2E1F
c 0317 0035 0030 001C 0018 0013 000C 000B 000B 0008 0004 0005 0003 0001 2C40
d 0300 0000 0000 0001 00FE 0001 0000 0000 0000 0000 0000 0000 0000 0000 0402
```

Werte 10 Bit benötigt wird. Einfache Lösung ist aus jeweils 32 Bit zwei 8 Bit Teilworte zu bilden und diese durch XOR zu verknüpfen (Bild 5).

Test

Man läßt den Generator wieder 1024 Muster erzeugen die dann über die Hash-Funktion auf 10 Bit Adressen komprimiert werden. Anhand dieser inkrementiert man jeweils einen von 1024 16 Bit Zählern die die Speicherzellen des EEPROMs darstellen. Im Idealfall enthält jeder Zähler nach Ende des Tests den Wert 1. Jede RFID-ID wäre dann einer anderen Adresse zugeordnet worden. Praktisch kommt es aber zu Kollisionen, also enthalten manche Zähler Werte grösser 1. Dementsprechend werden einige Adressen übehauptnicht angesprochen und enthalten den Wert 0.

Tabelle 1 zeigt die Ergebnisse für 8 Bit CRC, Fletcher und XOR. Mit Testdaten erzeugt vom Binärzähler und LFSR, letzterer mit 3 verschiedenen Startwerten (vgl auch Bild 4). Die Spalte „1“ enthält die guten Fälle, rechts davon sind die Kollisionsfehler. Um den Vergleich zu vereinfachen ist am Ende noch eine gewichtete Fehlersumme aufgeführt, die Mehrfachkollisionen verschärft bewertet. Dreifachkollisionen mal 2, Vierfachkollisionen mal 4, Fünffach mal 8 usw.

Die CRC erreicht zwar hier nahezu ideale Werte für den Zähler. Aber das Ergebnis von Fletcher ist ausgeglichener. XOR ist sehr schlecht, man beachte die Vielfach-Kollisionen weit rechts in der Tabelle.

[1] „PN-Sequenzen“ embedded (5) S. 6

[2] Jain „A Comparison of Hashing Schemes for Adress Lookup in Computer Networks“ IEEE Trans. on Com 1992/10 S. 1570 - 1573

Lineare Interpolation in Tabellen

Ein simples, wohlbekanntes Verfahren wird genauer wenn man die Tabellen optimiert.

Hier anhand der Berechnung

y = x^2,5

dargestellt. Wobei für x Werte von 0 ... 400h berechnet werden sollen. Eine direkte 32 Bit Tabelle würde 4kByte Speicher benötigen. Die hier gewählte Alternative ist, eine verkürzte Tabelle zu verwenden die nur jede 4. Stützstelle enthält und damit nur 1kByte belegt. Die Berechnung der 3 anderen Punkte durch lineare Interpolation (Bild 1) erfordert nur Additionen und Shiftbefehle und ist damit auch auf Controllern sehr leicht durchführbar. Dabei ist eine Sprungzieltabelle die von den untersten beiden Bits von x gesteuert wird eine geeignete Art der Implementierung (Listing LINT.F74)

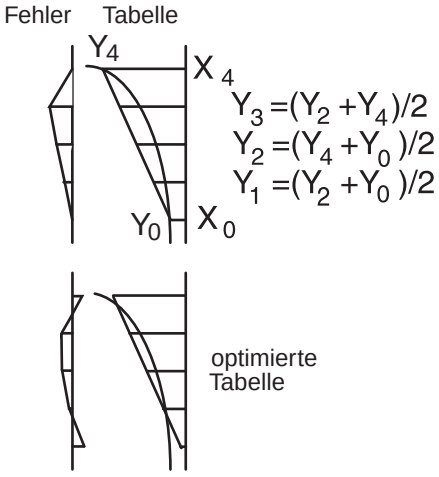


Bild 1: Interpolation, direkt & optimiert.

Tabelle 1: Fehler

unoptimierte Tabelle:

X	Ref Y	Fehler = Approx - Ref	
0000	0000 0000	0000	<-
0001	0000 0001	0007	
0002	0000 0006	000A	
0003	0000 0010	0008	
0004	0000 0020	0000	<-
0005	0000 0038	000D	
0006	0000 0058	0012	
0007	0000 0082	000D	
0008	0000 00B5	0000	<-
...			
03F8	01F6 0EFB	0000	<-
03F9	01F7 4B79	00B3	
03FA	01F8 886E	00EF	
03FB	01F9 C5DB	00B3	
03FC	01FB 03BF	0000	<-
03FD	01FC 421C	00B3	
03FE	01FD 80F0	00EF	
03FF	01FE C03C	00B3	
0400	0200 0000	0000	<-

optimierte Tabelle:

X	Ref Y	Fehler = Approx - Ref	
0000	0000 0000	+ 0000	<-
0001	0000 0001	+ 0005	
0002	0000 0006	+ 0007	
0003	0000 0010	+ 0003	
0004	0000 0020	- 0006	<-
0005	0000 0038	+ 0006	
0006	0000 0058	+ 000A	
0007	0000 0082	+ 0004	
0008	0000 00B5	- 000A	<-
03F8	01F6 0EFB	- 0077	<-
03F9	01F7 4B79	+ 003C	
03FA	01F8 886E	+ 0078	
03FB	01F9 C5DB	+ 003C	
03FC	01FB 03BF	- 0077	<-
03FD	01FC 421C	+ 003C	
03FE	01FD 80F0	+ 0078	
03FF	01FE C03C	+ 003C	
0400	0200 0000	- 0078	<-

Optimierung

Wie in Bild 1 auch dargestellt ist, sollte man nicht einfach die ausgedünnten Werte der 4kByte Tabelle verwenden. Man hätte dann zwar an den Stützstellen minimalen Fehler, aber an den interpolierten Punkten um so mehr Abweichung. Wünschenswerter ist eine gleichmäßig kleine Abweichnung an allen Punkten. Wie in Bild 1 dargestellt, dadurch erreichbar, daß man den Wert an den Stützstellen etwas verschiebt.

(Fortsetzung S.16)