DATALÆRE 3

MED

COMAL

OG

COMMODORE C-64

MASKINKODE

Bent Sehested

TUSE

Forord

Denne arbejdsbog er opstået efter tilskyndelse fra Langeskov Ungdomsskole. En del elever ville gerne videre med datamaten end spil og afskrivning af bladenes endeløse lister.
Efter samråd blev vi enige om, at styring måtte være et afsluttet emne. Samtidig ville vi have en indføring i maskinkode, programmering i assembler, brug af værktøj som assembler, simulator og monitor samt, hvis tiden tillod, lære at skrive simple pakker - og dermed hovedsagelig arbejde i COMAL.
Jeg har søgt at finde nogle almene emner frem. Hovedsigtet har været, at eleverne skulle lære nogle arbejdsmetoder og vises nogle ideer, der også kunne bruges på andre maskiner, - men alle øvelser sker naturligvis på vores C-64.
Da der ikke findes særlig megen litteratur om styring i COMAL og ingen om fremstilling af maskinkodepakker, har sigtet været at vise eksempler på styring af modeller, her opbygget i Fischertechnik og fremstilling af pakker, der kunne lette os programmeringen. Pakkerne findes i bogens 2.del.+) I teksten findes en del valgopgaver. Det er ikke meningen, at alle elever skal igennem alle øvelser. Valgopgaverne kan løses udfra den enkeltes interesse.
1.kursus har haft en varighed af 54 timer og det har givet en meget stram plan, der har nødvendiggjort at tekststykker blev forberedt som hjemmearbejde.
Bag i heftet findes en del tekniske sider (mærket i hexadecimal).
Ønsker man samarbejde med elektronik for at fremstille de beskrevne hjælpemidler, får man ekstra glæde af øvelserne.
Alle opgaver findes på den tilhørende programdiskette både som kildetekst (src.) og som objektkode (obj.).
Det er mit håb, at de valgte emner kan inspirere andre til at tage maskinkode op enten i valgfaget datalære (1990-) eller i ungdomsskolen.
+) I 2.del gennemgås fremstilling af pakker, styring af sprites, brug af interrupt, en simpel tekstbehandler, temperaturstyring og spil i maskinkode.

Bakkegårdsskolen, april 89

Bent Sehested

Bent Sehested

Indhold

Om data og talsystemer.................... side 1
Binære tal................................ - 5
Regning med binære tal.................... - 7
Hexadecimale tal.......................... - 9
Store og lille byte....................... - 10
Hukommelse C-64........................... - 12
Memory-map (tegning)...................... - 14
Om peek og poke........................... - 17
Mikroprocessor 6510....................... - 18
Den første maskinkode..................... - 22
Instruktionssættet for 6510............... - 24
Brug af indexregistre..................... - 27
Subrutiner................................ - 29
En simpel monitor i COMAL "se'lager"...... - 32
Enkeltskridt-simulator som pakke.......... - 32
Færdige subrutiner i kernal............... - 34
Brugerporten på c-64...................... - 35
Om bit-operationer........................ - 38
Statusregister og stakken................. - 43
Trafiklys i maskinkode.................... - 46
Om assemblerprogrammer.................... - 49
Assembler c-64 - vejledning............... - 50
Kildetekst og objektkode.................. - 54
Lav'data'linjer........................... - 56
Smon - vejledning for maskinkodemonitor... - 57
Tidsbestemmelse - registre................ - 61
Tællere og Fi-2 klokken................... - 65
Acceleration forsøg med lysbro............ - 66
Absolut x-indexeret adressering........... - 68
Relative spring........................... - 70
Morsetræner............................... - 71
Styring af motorer........................ - 76
Analog/digital konverter.................. - 80
Universalprint til motorstyring........... - 84
Fartregulering............................ - 86
Tekniske vejledninger fra side............ - $01

Datamater og talsystemer

Hvad er data ?

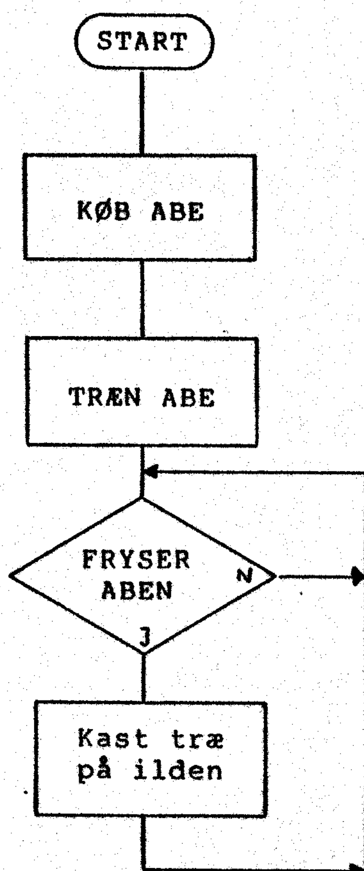
Her er nogle eksempler:

1. Madsen
2. 750 kr.
3. 07381444
4. 9c 12
5. mandag, kl. 13.00

1. er antagelig et efternavn
2. er en beløbsangivelse, - måske i danske kroner
3. ser lidt mærkelig ud, men cifre og tegn kan tyde på, at det er et telefonnummer.
4. på samme måde har dette udtryk i sig selv ingen betydning, men hvis du viser koden til bibliotekaren, kan han sige dig navnet på elev nr. 12 i 9.c
5. er en tidsangivelse.

Som det fremgår her, kan data i sig selv have flere betydninger; - og derfor er det vigtigt at definere data entydigt, når du skal i forbindelse med en datamat.

Hvad er en datamat ?



Kaptajn Haddocs farfar, der levede før automationens tidsalder, løste sit opvarmningsproblem ved at købe sig en abe.
Han trænede den til at kaste træ på ilden, hvis den begyndte at fryse.
Hændelsesforløbet kan i symbolsprog se ud som vist på diagrammet her ved siden af:

Hvis den samme opgave skulle løses idag, ville han nok anskaffe sig et oliefyr med en automatisk regulator.

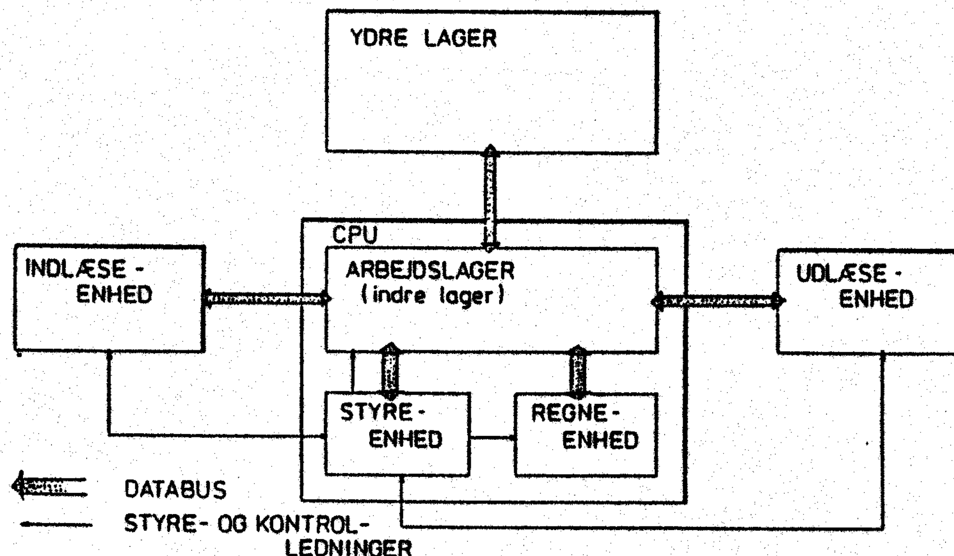
Ud fra dette eksempel, kan vi vel nok blive enige om, at vi bør skelne mellem automatiske apparater, der udfører deres funktioner automatisk - og de datamater, der først udfører automatiske funktioner efter at have gennemført en sammenligning eller en beregning.

Opgave: Hvilke af de følgende apparater rummer en datamat ?

1. En fuldautomatisk vaskemaskine.
2. En regnemaskine (lommeregner).
3. En elektrisk skrivemaskine.
4. En rumtermostat.
5. En kilomatertæller med speedomater.

Forklar hvorfor du er kommet til dit resultat.

HUSK: En datamat udfører sine ordre efter først at have foretaget en sammenligning eller en beregning.
Se skitsen, der stammer fra bog 1, side 14.



Input - Her foregår kommunikationen fra den ydre verden.
Output - Her forekommer resultater og meddelelser til omverdenen.
Lager - Her gemmes ordrer og data. (Arbejdslager).
Foruden er der et baggrundslager, f.eks. en diskettestation, der kan rumme programmer og data.

Kontrolenheden styrer hele processen ved hjælp af de ordrer, der hentes fra arbejdslageret.

Om forskellige datamattyper.

Datamaten kaldes analog eller digital alt efter hvordan den omdanner input til output.

Centralvarmeelementets termostat er en analog datamat. Målingerne (input) foregår jævnt og ventilens bevægelse (output) følger kontinuert. En lille ændring i temperaturen giver en lille ændring i ventilens stilling.

Regneenheden til et lysreguleringskryds er en digital datamat. Målingerne foregår jævnt (den måler evt. tiden.). Når bestemte værdier er opnået, skifter lamperne tilstand. Lamperne kan jo kun være enten tændte eller slukkede.

I datamatslang betegnes tændt og slukket ofte med "1" og "0". Denne skrivemåde vil vi vende tilbage til senere i kurset.

Nu behøver du ikke at vide, hvorledes datamaten er opbygget, men det øger din forståelse og mulighed for at benytte den, hvis du har en grundlæggende viden om dens virkemåde.

Foreløbig har vi omtalt begreberne:

data, datamat, databehandling, analog og digital.

Så skal vi igang med talsystemer.

Før du kan styre noget som helst, er der et par emner, du bør sætte dig ind i, idet kommunikationen med datamaten foregår ved hjælp af nogle talsymboler (maskinkode eller maskinsprog).
Da een datamattype kan være baseret på et talsystem, en anden på et andet, bør du blive fortrolig med talsystemer i almindelighed.

ROMERTAL
er et eksempel på et ikke-positionsafhængigt talsystem, idet værdien af et symbol ikke er bestemt af dets plads i tallet.

I	betyder	altid	1
V	-	-	5
X	-	-	10
L	-	-	50
C	-	-	100
D	-	-	500
M	-	-	1000

Det er nu muligt at kombinere de forskellige symboler:

IV betyder 4
VI - 6

En regel, der kan forklare dette lyder:

Et mindre tal foran et større, trækkes fra det større.
Ellers ordnes tallene efter faldende værdi.

XXVI betyder 10+10+5+1 = 26

Hvad betyder:

1) MDCCCCLXXXVI ?
2) MCMLXXXV ?

En af ulemperne ved romertal er de vanskelige regneoperationer.
Fosøg at lægge to tal sammen, f.eks.:

MDCCCL
CLXI
facit ______

kontroller med titalssystem:
facit ______

Kan du forklare, hvordan du klarede opgaven ?

Prøv at trække XII fra LX

Det er nok for os nu. Udregningerne er temmelig omstændelige, når vi er vant til at bruge 10-talsystemet, men nu ved du, hvad systemet rummer.

Se evt. programmet, der omdanner ti-tal til romertal. Det findes på disketten.

load "romertal1" <RETURN>

Øvelse: Fremstil et program, der laver den omvendte operation.

Det ligger måske allerede i luften, at vi bør benytte os af et positionsafhængigt talsystem. Her afgøres tegnets værdi af dets position i tallet som et hele.

DECIMALE TAL

Det DECIMALE talsystem skal bruges som eksempel på et system, hvor tegnets plads i tallet bestemmer værdien. Vi vælger at vise 10--talsystemet, fordi vi alle har træning i at anvende det.
Værdier angives af de 10 tegn 0-9 og den yderste plads til højre er enernes plads; den næstyderste tiernes plads o.s.f.

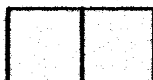
eks. 38756 3 8 7 5 6

6	6x	enere =	6
5	5x	tiere =	50
7	7x	hundrede =	700
8	8x	tusinde =	8000
3	3x	titusinde =	30000
			38756

Flytter du et komma (som ikke ses i eksemplet) 1 plads til højre, svarer det til, at du ganger med 10; - flyttes kommaet 1 plads til venstre, deles med 10.
Dette talsystem giver lette regneregler, antagelig fordi vi er vant til at bruge det. Der kan opbygges mange positionsafhængige talsystemer og det er iblandt dem vi skal finde et system, der egner sig til brug i en digital automat.
Ethvert positionsafhængigt talsystem har et grundtal (en base), der samtidig er antallet af forskellige værdier, værdien nul "0" medregnet, som kan anvendes i en vilkårlig position i et tal. I eksemplet herover kan alle værdier fra 0 til 9 bruges. Grundtallet er altså 10 for et decimalt tal.

BINÆRE TAL.

Hvis vi vælger et talsystem med grundtal 10 i datamaten, løber vi ind i en masse besværligheder. I datamatens kredsløb anvendes elektriske strømme og spændinger. Forestil dig, at vi skulle have strømme til at løbe med 10 forskellige farter som et udtryk for de 10 anvendelige tegn. Det ville hurtigt blive kaos. Det er lettere at opfatte en transistor som en kontakt, og så er der kun 2 mulige stillinger: lukket og helt åben. Ved at vælge er talsystem med base 2, er der mulighed for en yderst præsis funktion. Vi forkaster det decimale system og vælger det binære (et talsystem med base 2).
Hvad hedder de symboler, som base 2-systemet kan bruge i hver position ?



Selvfølgelig 1 og 0, som svarer til transistorens åbne (1) og lukkede (0) tilstand. En position i dette system benævnet en BIT.
Ofte får vi brug for omregning mellem binære tal og decimale tal.
Hvis det skal foregå pr. håndkraft, kan vi tænke os følgende:

```
position  3  2  1  0               position  3  2  1  0
tital     5  2  8  7               total     1  0  1  1
          |  |  |  |                         |  |  |  |
          |  |  |  L 7x10⁰     7             |  |  |  L 1x2⁰   1
          |  |  L--- 8x10¹    80             |  |  L---- 1x2¹   2
          |  L------ 2x10²   200             |  L------- 0x2²   0
          L--------- 5x10³  5000             L---------- 1x2³   8
                            ----                               ---
                            5287                                11
```

Prøv selv at omsætte følgende 3 binære tal til decimal:

1) 01001101 =
2) 00001100 =
3) 11111111 =

Bemærk serien af vægte; de vil ofte dukke op:

potens	n	7	6	5	4	3	2	1	0
vægt	2	128	64	32	16	8	4	2	1

Skal vi gå den modsatte vej, kan det foregå sådan:

Omsæt decimal 223 til binært tal.

```
                                             res$:=
tal:=223;        er der en "128"er           hvis ja   1   ellers   0
res$:=1
tal:=tal-128;    er der en  "64"-               -  ja  1     -      0
res$:=11
tal:=tal-64;     er der en  "32"-               -  ja  1     -      0
res$:=110
tal:=tal-0 ;     er der en  "16"-               -  ja  1     -      0
res$:=1101
tal:=tal-16;     er der en   "8"-               -  ja  1     -      0
res$:=11011
tal
.
.
res$:=11011111
```

1) Omsæt decimal 41 til binærtal
2) Omsæt decimal 239 - -

Her har du en funktion, der kan lave det samme. Fremstil et program, der som input har et decimalt tal; gemmer det i variablen tal; kalder funktionen og som output giver det samme tal i binær form. Placer input og output pænt på skærmen med lidt vejledende tekst. Funktionen kan f.eks. kaldes sådan:

```
CURSOR 12,5
PRINT dec'bin(tal)
```

Denne programstump skal senere indgå i et program, der skal omregne fra binær til decimal og omvendt. Når programmet fungerer efter hensigten, bør du save det på diskette. Find selv et navn. Foretrækker du at liste funktionen som en ASCII-fil, må du følge konventionerne for programnavne og bruge kommandoen:

```
list dec'bin "lst.dec'bin" <return>
```

På den måde er det muligt at merge funktionen til et program senere.
Skal man den anden vej, altså lave binær til decimal, ville det

REGNING MED BINÆRE TAL.

Opgave 1) Hvad er decimalværdien af "11111100" ?
2) Hvad er den binære betegnelse for 251 ?
3) Konverter 19 til binær og tilbage igen.

Det er godt at kunne gå fra binær til decimal notation, men det bør også være muligt at regne med binære tal.
Det letteste er at addere. Her er reglerne:

```
0 + 0 =    0
1 + 0 =    1
0 + 1 =    1
1 + 1 = (1)0
```

Tegnet i parantes betyder en mente, der overføres til næste bit.

```
Eksempel:   2     10      5    101      9   001001
          + 1   + 01    + 2  + 010   + 22 + 010110
            3     11      7    111     31   011111
```

4) Beregn selv 5 + 10 binært.

I eksemplerne var ingen mente. Klarede du opgave nr. 4 ?

```
   3   0011      7   0111      27   011011
 + 1 + 0001    + 3 + 0011    + 23 + 010111
   4   0100     10   1010      50   110010
```

5) Kan denne opgave klares med 4 bit ? 15 + 1
6) Hvor store tal kan behandles med 8 bit ?

Den viste metode kan behandle positive hele tal og 1 byte kan rumme op til 255, men det er ikke alle tal, der er positive. Vi laver en aftale om, at bitten længst til venstre skal vise tallets fortegn.

BINÆRE TAL MED FORTEGN.

"0" betyder et positivt tal; "1" et negativt tal. 11111111 vil efter denne metode betyde -127; 01111111 vil betyde +127. Vi får mulighed for at vise fortegn, men tallets max. størrelse skrumper ind til 127.

7) Hvordan vil du skrive -5 binært med fortegn ?

Vi kan nu vise en talrække fra -127 til +127, men vi kan ikke regne med tallene. Før det kan lykkes, må du præsenteres for begreberne etkomplement og tokomplement. Ved udtrykket komplement til 8 menes -8.

ETKOMPLEMENT og TOKOMPLEMENT.

I denne skriveform betegnes alle positive tal i alm. binært format. 7 skrives 00000111. Komplementet -7 laves ved at ombytte alle bit i det positive udtryk. -7 er altså 11111000.

```
binært  13 skrives 00001101
       -13    -    11110010
```

Venstre bit angiver stadig fortegn. Af etkomplement laves tokomplementet ved at addere 1.

```
Eksempel:  Fremstil -3 i tokomplement.
           3 skrives              00000011
           etkomplement           11111100
                                  +      1
           -3 i tokomplement      11111101
```

Nu prøver vi en beregning: 7-5 skal give +2

```
   7      00000111        5 skrives     00000101
  -5    + 11111011        etkomplement  11111010
  --    ----------                      +      1
  +2      00000010        tokomplement  11111011
```

En opgave, hvor facit er negativt: 5-7 skal give -2

```
   5      00000101        7 skrives     00000111
  -7    + 11111001        etkomplement  11111000
  --    ----------                      +      1
  -2      11111110        tokomplement  11111001
```

på bit 7 ses, at facit er negativt. Så mangler vi at vise, at udtrykket har værdien 2. Det kan klares ved at tage tokomplementet:

```
11111110
00000001
+      1
00000010
```

Alment gælder, at subtraktion kan klares ved addition af tokomplementet. Ved at udtrykke negative tal som tokomplement, kan vi anvende de almindelige regler for binær addition til at klare både addition og subtraktion.

8) Udregn tokomplementet af 20 og udregn derefter tokomplementet af resultatet.
   Er slutresultatet det oprindelige tal 20 ?

MENTE og OVERLØB.

I det følgende eksempel opstår en fejl, idet microprocessoren kun viser 8 bit:

```
    128     10000000
  + 129     10000001
         -------------
         (1)00000001,
```

hvor (1) betyder mente. Denne mente må vi behandle specielt, hvilket vi vil se, når vi præsenteres for microprocessorens opbygning.
Herunder findes en tabel, der viser binærkoden for tokomplementet.
Forsøg at beregne værdierne for binær 4, 5, 6, 7, 8 og binær -3, -4, -5, -6, -7, -8. Kontroller med de indskrevne værdier.
Denne tabel får du brug for, når du skal beregne spring i programmerne.
Søjlen med hexkode behøver du ikke at tænke over nu. Den vil vi benytte senere, men det er praktisk at den findes i den samme tabel som binærkoden.

decimal	binærkode for tokomplement	hexkode
+127	01111111	7f
+126	01111110	7e
+125	01111101	7d
+124	01111100	7c
+123	01111011	7b
.		
.		
.		
+ 11	00001011	0b
+ 10	00001010	0a
+ 9	00001001	09
+ 8		
+ 7		
+ 6		
+ 5		
+ 4		
+ 3	00000011	03
+ 2	00000010	02
+ 1	00000001	01
0	00000000	00
- 1	11111111	ff
- 2	11111110	fe
- 3	11111101	fd
- 4		
- 5		
- 6		
- 7		
- 8		
- 9		
- 10	11110110	f6
- 11	11110101	f5
.		
.		
.		
-126	10000010	82
-127	10000001	81
-128	10000000	80

Det må understreges, at datamaten kun forstår binære koder. Alle input og ordrer må sendes i denne kode. Programlinjer i COMAL eller BASIC oversættes før udførelsen og resultatet oversættes igen fra maskinkode til noget forståeligt, før det vises på skærm eller printer. Disse oversættelser frem og tilbage er en af de væsentlige grunde til den store tidsmæssige forskel mellem maskinkode og programafvikling i højere sprog.
Skulle vi skrive maskinkode i binær form, blev resultatet hurtigt uoverskueligt. De mange 0 og 1 ville løbe sammen. Derfor har man valgt at omskrive binærkoden til en anden notation, som datamaten kan oversætte og forstå.

HEXADECIMALE TAL.

De 8 bit deles i to blokke(nibbels) med 4 tegn i hver. Hvor mange tegn kan en nibbel rumme ? - Så konstruerer man et talsystem med base 16 og kan nu angive de binære 8 bit med 2 tegn i hexadecimal notation. Til daglig kaldes disse tal for hex-tal. Vi anvender de 10 tegn fra 0 til 9 som vanligt, men vælger at benytte bogstaver for de sidste 6 tegn. Sammenhængen fremgår af dette skema:

decimal	binær	hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Nu har vi 3 talsystemer at benytte. Derfor må vi kunne kende dem fra hverandre. Datamaten kan arbejde med dem alle i COMAL og følgende konvention er vedtaget:

decimal	binær	hexadecimal
319	%010011	$3f

Decimale tal har ingen tegn foran. Alle binære tal kendes på % og alle hexadecimale har et $ foran.

9) Hvad er 15 i hextal ?
   Hvad er %1101 i hextal ?

Hvordan kan vi lave et program, der omsætter fra decimal til hexkode ? Starter vi med området 0-255, må det kunne klares som vist her:

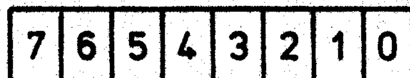
lovlige tegn er "0123456789abcdef"
tal delt med 16; antal gange giver 1. ciffer
tal delt med 16; divisionsrest giver 2. ciffer
disse to cifre skal omsættes til lovlige tegn,
hvor 0 indtager 1.plads.

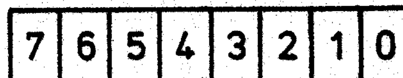
I COMAL kan det se således ud:

```
FUNC hex$(n) CLOSED
  DIM hexciffer$ of 16, resultat$ of 4
  hexciffer$:="0123456789abcdef"
  resultat$:=hexciffer$(n DIV 16+1)
  resultat$:+hexciffer$(n MOD 16+1)
  return "$"+resultat$
ENDFUNC hex$
```

STORE BYTE, LILLE BYTE.

Vores datamat har mere end 255 lagerpladser. Da hver lagerplads (celle) kendetegnes ved at nummer (en adresse), må der opstå problemer, hvis vor talverden stopper ved 255. Hvordan klares dataværdier, der er større end $ff ?
Vi må bruge 2 byte til at angive adressen, altså hægte to 8-bit tal i forlængelse af hinanden. Se tegningen på næste side.





store byte lille byte

Med 2 byte er der $2^{\uparrow 16}$ kombinationsmuligheder, altså 65536 pladser.

$\$a90f = 10*16^3 + 9*16^2 + 0*16^1 + 15*16^0$

Hvad bliver det forresten decimalt ?

Med hex-tal kan vi angive ret store tal med 4 pladser. $ffff angiver hermed 65536 i 10-talsystemet.

Funktionen ovenfor må hermed udvides, så den kan klare hex-tal, opbygget af store byte/lille byte.

```
FUNC hex$(n) CLOSED
  DIM hexciffer$ of 16, resultat$ of 12
  hexciffer$:="0123456789abcdef"
  store:=n DIV 256; lille:=n MOD 256
  resultat$:=hexciffer$(store DIV 16+1)
  resultat$:+hexciffer$(store MOD 16+1)
  resultat$:+hexciffer$(lille DIV 16+1)
  resultat$:+hexciffer$(lille MOD 16+1)
  return "$"+resultat$
ENDFUNC hex$
```

Se evt. programmet med omregningerne. Det ligger på disketten under navnet "omregn bin'hex".

- Og så er det tiden at gå igang med fremstilling af stik til userport og pladen med de 8 lysdioder. Se tegningerne bag i heftet.

DATAMATENS HUKOMMELSE.

Datamaten skal bruge lagerplads, hvor program og data kan lægges. Microprocessoren behøver lagerplads til mellemresultater under arbejdet og værdier, der skal bearbejdes senere, skal gemmes så regneenheden kan finde dem igen. Denne hukommelse kan opfattes som en lang lagerbygning med en sidegang. Reoler er stillet op bag hinanden, så lagermanden kan komme til alle reoler ad denne sidegang. Hver reol har et nummer og på hver reol kan ligge 1 byte. Reolens nummer kaldes også for reolens adresse. Når adressen opgives i hex-tal, må det betyde, at vi skal bruge 2 byte for at kunne nummerere alle hylder (celler), idet der totalt er 65536 hylder i C-64.

Adresse: store byte*256+lille byte.

De første 256 celler kan adresseres ved brug af kun lille byte. Store byte er $00. De næste 256 celler har store byte $01; tredie gang vi gennemløber værdierne i lille byte er store byte $02 o. s. fr.
Denne skrivemåde har delt adresserne i sider. Vi taler om at adresserne fra 0-255 ligger på side 0 (zero-page); næste gennemløb af lille byte giver adresser på side 1, tredie gennemløb af lille byte giver adresser på side 2 o.s.v. - Ialt er lageret bygget op af 256 sider med hver 256 celler, ialt de omtalte 65536 celler.

Regner vi i maskinskrevne A-4 sider (50 linjer a 60 anslag) ... betyder en side tekst 3000 anslag og så kan datamaten fyldes op af 21 sider maskinskrevet tekst. Nu er al denne plads ikke til vores rådighed for program eller data. Tænder vi C-64 med COMAL-kapsel isat, melder den sig med:

$$$ Commodore C-64 Comal 80 rev. 2.01 $$$

og angiver straks, at der er 30.714 bytes fri til vores brug. RAM betyder Random Access Memory @:det lager, hvor vi kan lægge program og data og hvorfra vi kan læse igen. Ved at skrive ordren:

```
size <return>     vises

prog    data    free
00000   00000   30714
```

På skærmen vises, hvor megen lagerplads der er ledig. Når ikke alle lagerpladser er fri når datamaten tændes, må der ligge noget i lageret allerede fra start - og det er også tilfældet.
Foruden det RAM-lager vi kan anvende til programmer, findes flere ROM-lagre (Read Only Memory) - lagerpladser hvorfra vi kan læse, men ikke skrive til. Indholdet i et ROM-lager går ikke tabt, når datamaten slukkes. I disse ROM-lagre opbevares de oplysninger, datamaten skal benytte i startfasen for at kunne vise billede på skærmen og være klar til forbindelsen til tastatur og diskettestation. Den skal også kunne hente et program ind, lægge det på plads og gøre klar til afviklingen.

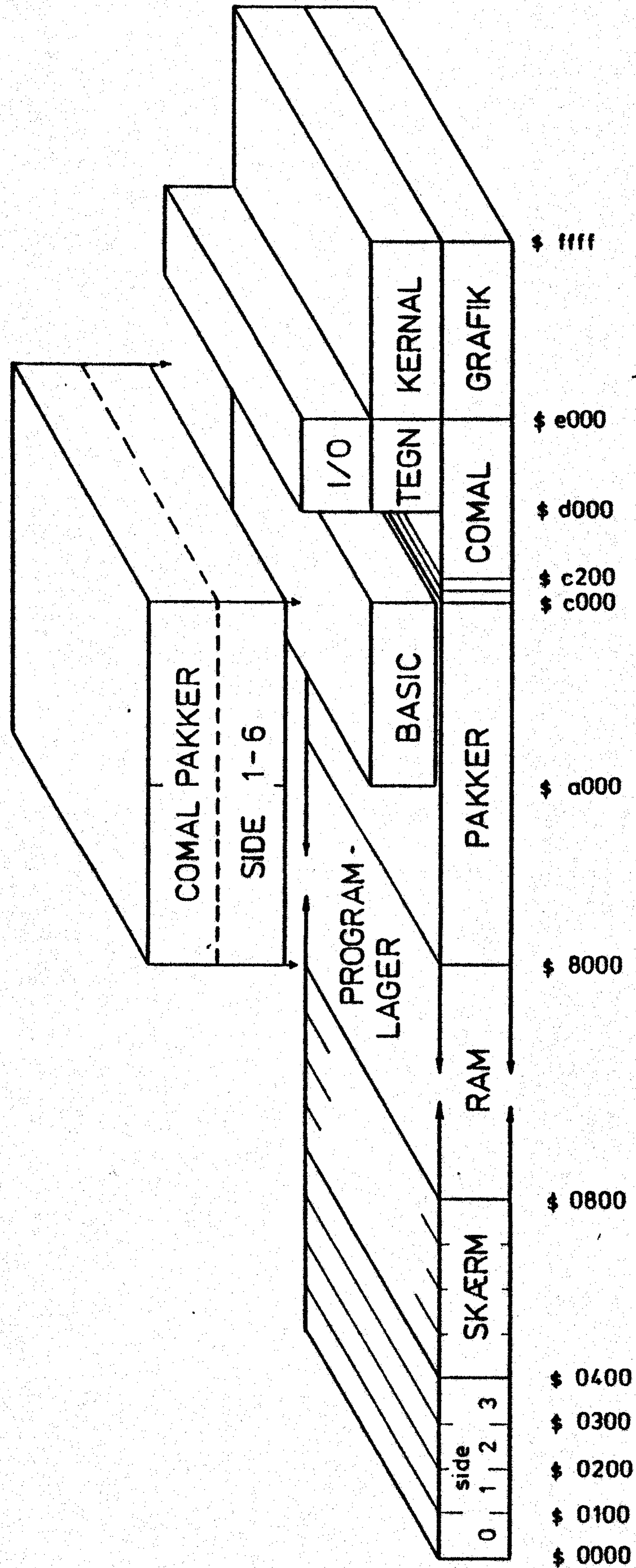
På tegningen næste side ser du den omtalte lagerbygning med adresserne antydet.

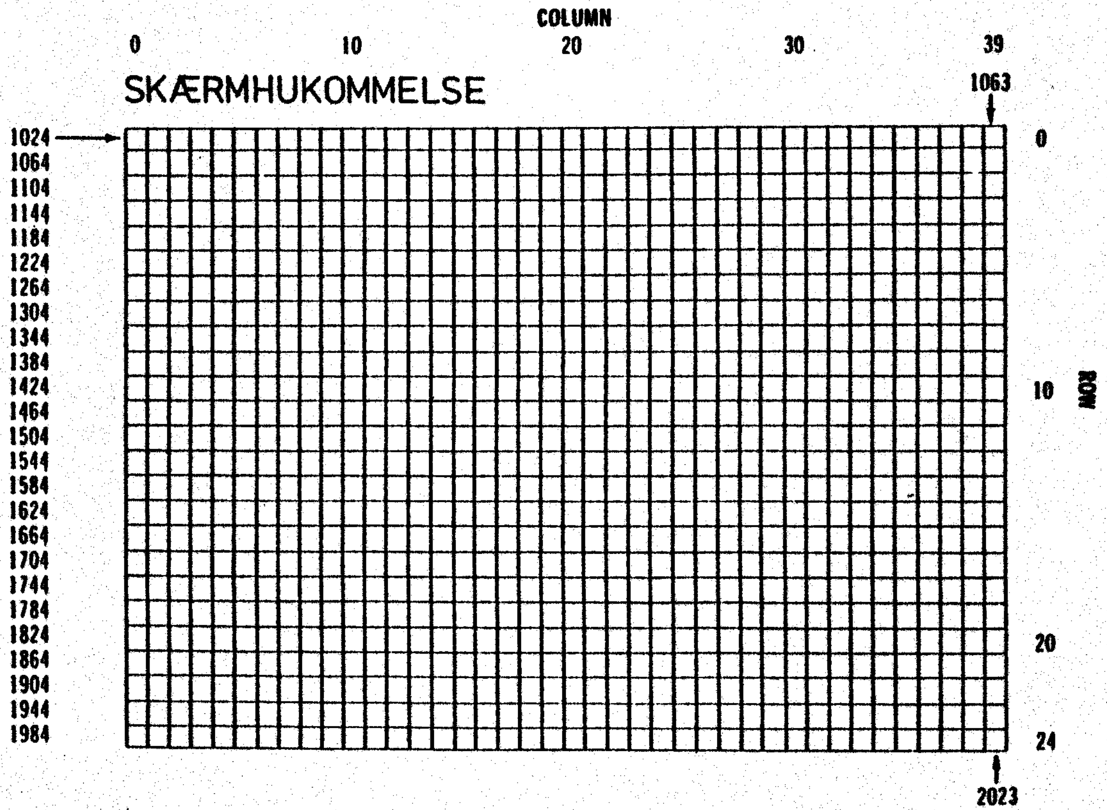
side opgave
$00 Her findes pegepinde til datamatens egen husholdning. Se bem. nedenfor.
$01 System-stack - holder orden på delopgaver under programmets afvikling.
$02 System-vektorer - pegepinde, der bruges under pro gramafviklingen.
$03 System-vektorer + kassettebuffer, se bem.
$04-$07 RAM Skærmhukommelse. Her har hver plads på skærmen sin adresse.
$08-$7f RAM for COMAL-programmer
Det er i dette område vi lægger programmer, der hentes ind med load, enter eller merge. Det er altså det almindelige arbejdsområde.
$80-$bf RAM som COMAL benytter. Her lægges de faste pakker, der gøres kendte med USE. Det er også muligt at lægge selvfremstillede pakker samme sted. Se bem.
$c0-$df Her ligger COMAL i en ROM. Comal hentes ind ved datamatens start. Her ligger systemvariable og herfra styres standardpakkerne.
$d0-$df Her findes en ROM med standardtegn og samtidig en ROM, der muliggør kontakten med omverdenen. Farvehukommelsen findes også i dette område som en RAM. Indkobling af disse dele sker efter tur og denne opgave styres af COMAL.
$e0-$ff Lageret har to etager. Nederst ligger det RAM-lager, der bruges til grafik. Derfor flyttes sprites hertil, når de vises på skærmen. KERNAL ovenover er et ROM-lager. Her ligger en masse færdige rutiner, der kan bruges af COMAL. Der findes en springtabel, der leder hen til den ønskede rutine.
$a0-$bf Her ligger en ROM med BASIC. Denne del er slået fra, når COMAL starter, men den kan hentes ind af COMAL til specialopgaver og den kan forlades igen efter programafviklingen med maskinkoder.

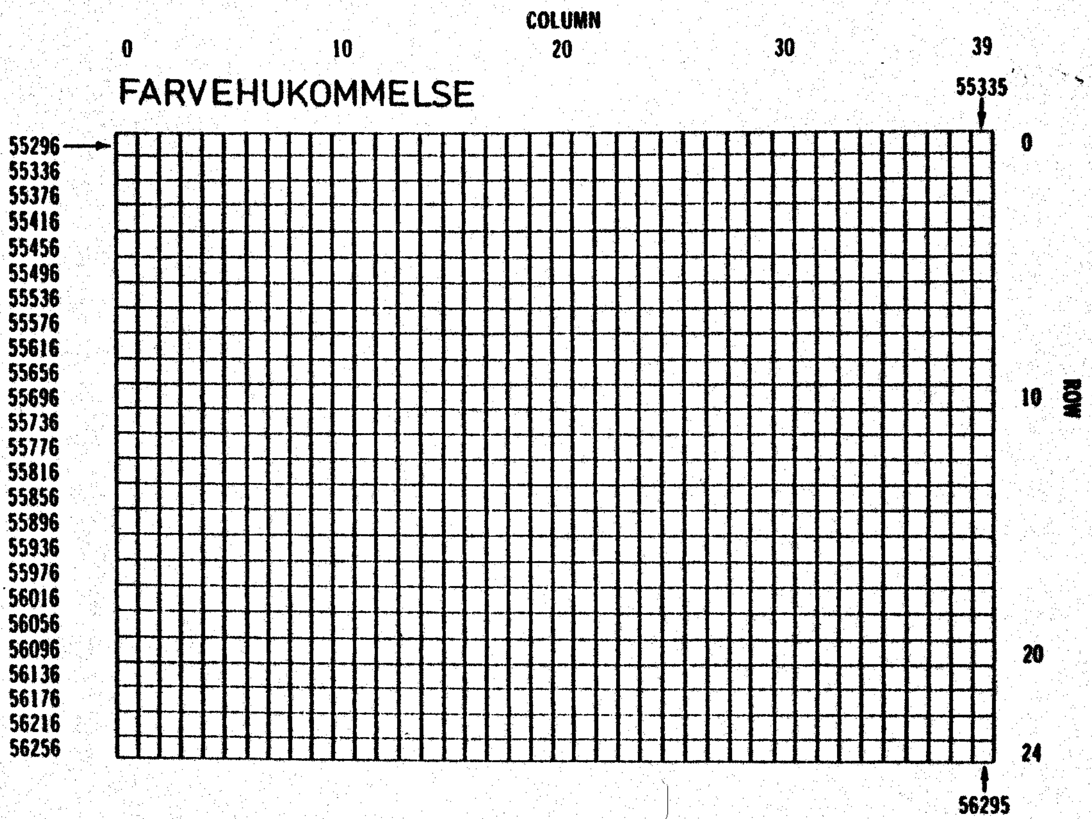
Bemærkninger til denne oversigt (Memory Map)

side $00 er der 6 tomme pladser, idet COMAL ikke benytter $55 og $fb-$ff.
side $01 er adgang forbudt
side $02 gælder det samme, og
side $03 findes en kassettebuffer, der ikke er i brug medmindre der er sluttet en datasette til maskinen. Denne buffer går fra $033c til $03ff, ialt 196 lagerpladser. Nogle COMAL-programmer benytter bufferens første 30 bytes til status$. For at komme udenom dette problem, bør du arbejde i det garanteret ledige område $035a til $03ff - altså 166 bytes. Pladsen er velegnet til små maskinkoderutiner, der ikke skal slettes med NEW.
side $04-$07 er en RAM, hvor skærmhukommelsen bruger 1000 celler. De sidste 24 er anvendelige til andre formål (se tavle)
side $c0 Her ligger en buffer for RS 232 input og
side $c1 rummer en tilsvarende buffer for output.
Disse to sider kan bruges i alle vore eksempler. Programmer lagt her vil ikke blive slettet med NEW.

C-64 COMAL MEMORY MAP




COLUMN
0
10
20
30
39
SKÆRMHUKOMMELSE
1063
1024
1064
1104
1144
1184
1224
1264
1304
1344
1384
1424
1464
1504
1544
1584
1624
1664
1704
1744
1784
1824
1864
1904
1944
1984
0
10
20
24
ROW
2023


COLUMN
0
10
20
30
39
FARVEHUKOMMELSE
55335
55296
55336
55376
55416
55456
55496
55536
55576
55616
55656
55696
55736
55776
55816
55856
55896
55936
55976
56016
56056
56096
56136
56176
56216
56256
0
10
20
24
ROW
56295

Hvor er der ledig plads ?

Side $00	$55, $fb-$ff	6	pladser
side $03	$035a-$03ff	166	pladser
side $07	$07f8-$07ff	24	pladser
side $c0	hele siden	256	pladser
side $c1	hele siden	256	pladser

Hvis maskinkodeprogrammer er større end de 512 bytes, der kan ligge på side $c0 og $c1, bør de formes som en pakke og dermed lægges i det ledige område fra $80 til $bf. Om fremstilling af pakker: se senere i heftet.

Efter tegningen af C-64 hukommelse, følger en side med skærmhukommelse, farvehukommelse og her er en tabel over farvekoder.

Farvekoder:

Skærmkode	Farve	ASCII-kode
$00	sort	144
$01	hvid	5
$02	rød	28
$03	turkis	159
$04	violet	156
$05	grøn	30
$06	blå	31
$07	gul	158
$08	orange	129
$09	brun	149
$0a	lyserød	150
$0b	mørkegrå	151
$0c	grå	152
$0d	lysegrøn	153
$0e	lyseblå	154
$0f	lysegrå	155

Øvelse: Hent programmet "omregn bin'hex" ind fra disketten. Start det og undersøg virkemåden ved at gennemføre nogle omregninger. Når du er fortrolig med programmet, kan du liste det ud på printer. Undersøg programmet ved at lave nogle gennemløb på udlistningen.

I I/U blokken findes en del adresser, vi vil få brug for under de forskellige opgaver.

Hex	Decimal	Opgave
$e000		
.		
.		
$dd0f	56591	kontrolregister, tæller B
$dd0e	56590	- , tæller A
.		
$dd07	56583	store byte, tæller B
$dd06	56582	lille byte, tæller B
$dd05	56581	store byte, tæller A
$dd04	56580	lille byte, tæller A
$dd03	56579	dataretningsregister B
$dd02	56578	- A (kun 3.bit)
$dd01	56577	dataregister B
$dd00	56576	- A

```
        .
        .
$dbff      56319
        .              farvehukommelse
$d800      55296
        .
$d41a      54298       spilport 2, pot y
$d419      54287          -     1, pot x
        .
$d021      53281       farve tekstskærm
$d020      53280       farve skærmens ramme (border)
$d018      53272       kontrolreg. karaktersæt
```

OM POKE og PEEK

Find side 269-272 i COMAL-håndbogen eller side 376-77 i programmers ref. guide. Her kan du i tabeller se skærmkoder for bogstaver og for farve. Når du samtidig har oplysninger om lagerets opbygning, er tiden inde til et lille eksperiment. Du kan lægge en talværdi direkte ind på en adresse med kommandoen:

POKE adresse,værdi <RETURN>

Poke kan nærmest oversættes ved "stød ind". Adresse er adressen på den lagercelle, hvor man vil lægge talværdien "værdi".
"o" har skærmkode 15; skærmhukommelsens 1. plads har adresse 1024.
Hvis du skal lægge et o i skærmens øverste venstre hjørne, kan det ske med kommandoen:

POKE 1024,15 <RETURN>

Øvelse: Undersøg hvad der sker, hvis du laver den samme opgave, men bruger hex-tal. - Kan du også lave skriften hvid ? - Du skal bruge farvehukommelsen.

Find skærmkoderne for bogstaverne i dit for- og efternavn og skriv dit navn på øverste linje på skærmen - midtfor !

Nu ligger der værdier i nogle af cellerne. Det ville være ønskeligt, om vi kunne kikke efter, om det var de rigtige værdier. Det gøres med kommandoen:

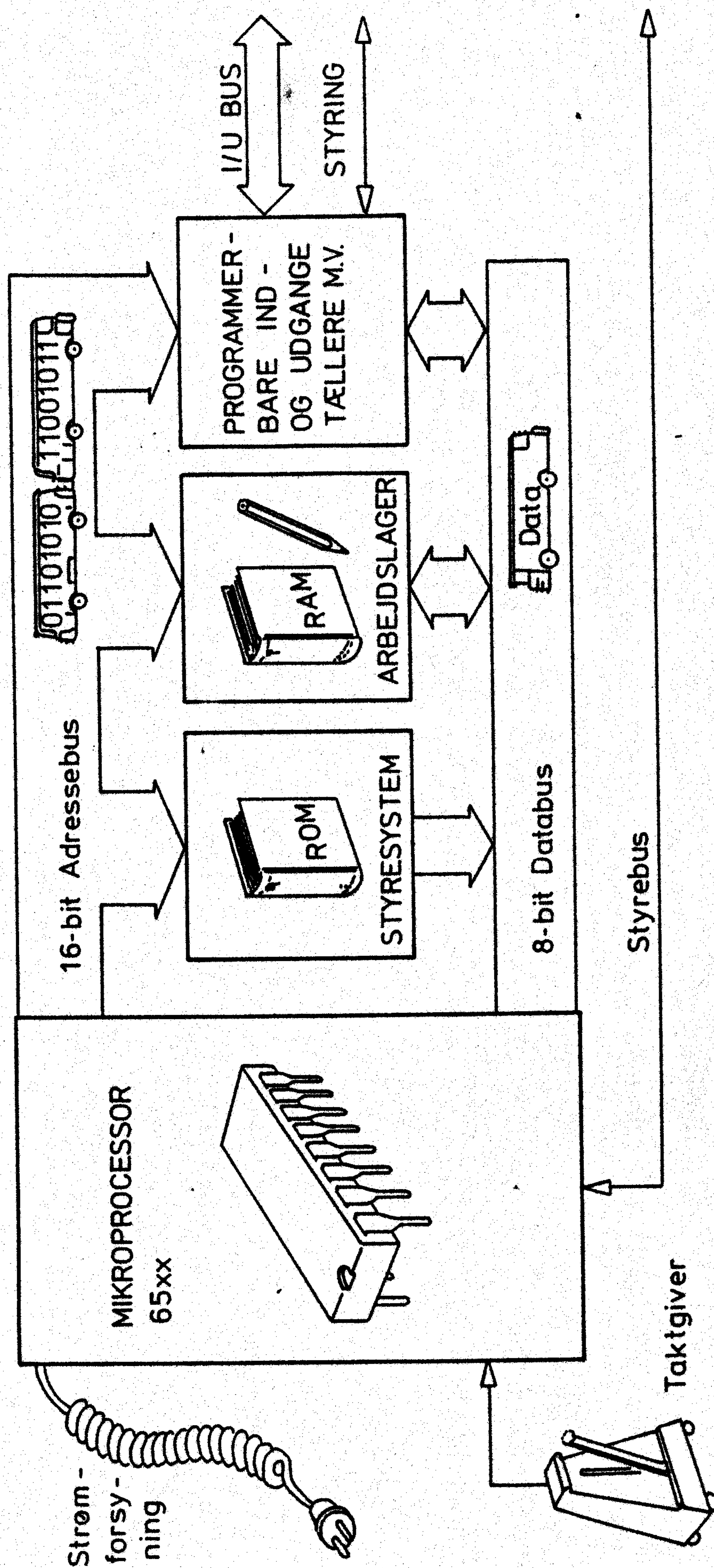
PEEK(adresse) <RETURN>

Skal det vises på skærmen, må der stå PRINT foran kommandoen. Peek kan oversættes med "kik ind" på adressen.

Øvelse: Undersøg et par af tegnene i datamatens opstartbillede. Ligger tegnene i de korrekte adresser ?

Ved kommandoen POKE kan værdier lægges i bestemte celler.
  -      -   PEEK spørger man til indholdet i en celle.
Begge kommandoer kan også bruges i programmer.

Dette er en langsommelig metode at POKE på. Inden længe skal du se hvordan vi kan skrive et program, så datamaten selv overtager denne opgave. - Før det sker, må du præsenteres for microprocessoren i datamaten. Den hører til familien 65xx og vores eksemplar er type 6510.


I/U BUS
STYRING
PROGRAMMER-
BARE IND-
OG UDGANGE
TÆLLERE M.V.
01101010
11001011
RAM
ARBEJDSLAGER
Data
16-bit Adressebus
ROM
STYRESYSTEM
8-bit Databus
Styrebus
MIKROPROCESSOR
65xx
Strøm-
forsy-
ning
Taktgiver

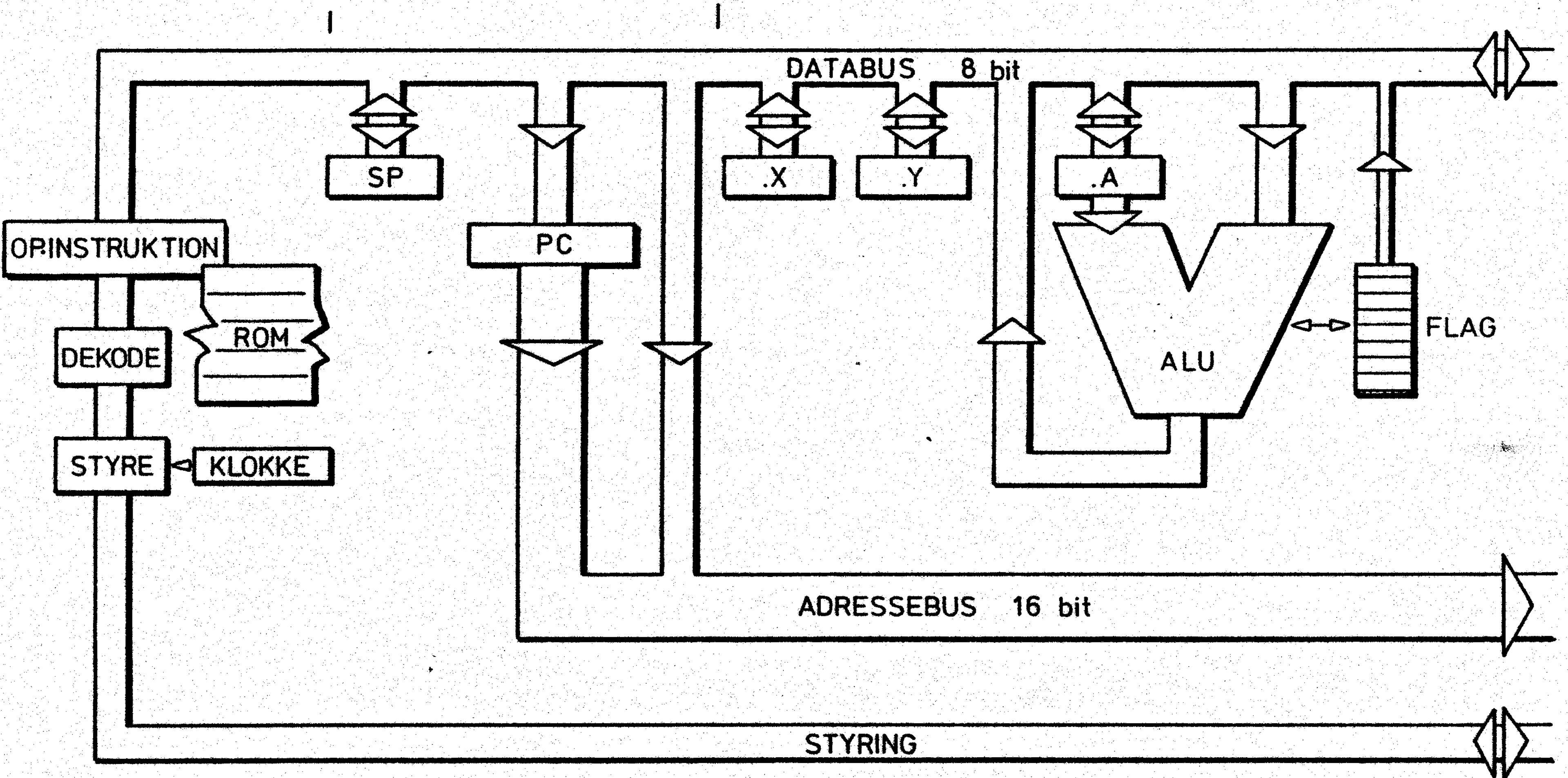
Denne enhed, der ofte ses omtalt som MPU (Micro Processor Unit), er udført som en integreret kreds, der har egne registre (lagerpladser), regneområde og arbejdslagre. Derforuden er der følgende komponenter:

ROM'en (Read Only Memory) indeholder systemets program.
RAM'en (Random Access Memory) er læse- og skrivehukommelse.
I/U antyder de grænseflader systemet skal have for at komme i forbindelse med omverdenen.

Tegningen på forrige side viser MPU'en til venstre. Herfra styres datatrafikken mellem de 65536 lagerpladser i hukommelsen. Dens styrke ligger i den fart, hvormed de enkelte operationer udføres. Via adressebus'en udvælges den lagercelle, hvorfra der skal hentes eller lægges data. Den har forbindelse til de faste rutiner i ROM og de frie lagerpladser i RAM. Ofte har man brug for en forbindelse ud af datamaten og disse forbindelser må også kunne nås. Derfor opdeles adressen i lille byte/store byte og overføres via 16 bit bus'en.
I indgangs/udgangsenheden ligger bl.a. styring af skærmbilledet, input fra tastatur og styring af de forskellige porte, der bruges: datasette, printer, diskettestation, userport, COMAL-kapsel m.v.
Databus'en overfører data mellem systemets enkelte elementer. Det vil typisk være transport af data mellem hukommelse og MPU.
Styrebus'en benyttes til omstilling mellem ind- og udgang fra de enkelte komponenter og mange gange til klarmelding, når en opgave er løst.
Alle disse operationer må være nøje tilpasset hinanden. Derfor fastlægger en intern taktgiver en arbejdsfrekvens på 980 MHz, d.v.s. omkring 1 million taktslag pr. sek. Nogle operationer bruger 2 taktslag, nogle 1, andre 3 eller 4 taktslag. Forestil dig en romersk gallej med 3 etager roere. Hvis disse roere ikke arbejdede i takt, ville al deres energi sikkert blive brugt til at undgå sammenstød mellem årerne indbyrdes. En særlig betroet slave blev anbragt på roernes dæk med den opgave at slå takten alle skulle arbejde efter; - han var gallejens interne taktgiver.

Den indre opbygning af 6510

På næste side ses et skema, der viser de forskellige dele af 6510. Det er vigtigt at forstå samvirket mellem dem for at kunne skrive maskinkodeprogrammer.
På tegningen ses ALU'en (Aritmetisk-Logisk Unit) som et "V". Her udføres sammenligning, addition eller logiske operationer mellem to tal, der anbringes på hvert sit af ALU'ens to ben. Når regneoperationen er udført, findes resultatet i bunden af ALU'en. Til enheden er akkumulatoren (.a-reg. eller kort .a) fast tilknyttet. Ved aritmetiske eller logiske operationer, vil indholdet i .a være den ene operand, mens den anden typisk vil være indholdet i en lagercelle i hukommelsen. Resultatet opbevares i .a. Denne arbejdsgang giver korte instruktioner.
Til højre for ALU'en findes et 8 bit statusregister, hvor hver bit bearbejdes for sig. Her sættes eller lægges flag, der skal signalere, om bestemte tilstande er indtruffet.
Af disse status-flag får vi først brug for N-, Z- og C-bittene.



MIKROPROCESSOR 65xx

N	-	,		Z	C

N:neg. N-flaget sættes (N-bit=1) hvis res. i ALU er negativt.
Z:zero Z-flaget sættes (Z-bit=1) hvis seneste operation gav 0
C:carry C-flaget sættes (C-bit=1) hvis 2 8-bit tal sammenlægges og resultatet ikke kan være i et 8 bit register (mente).

Carry anvendes meget ved talbehandling. De 3 status-bit aktiveres uden vores medvirken, men vi bruger dem for at teste, om bestemte situationer er indtruffet.

Et stykke til venstre på tegningen findes programtælleren (Program Counter). Den er et 16-bit register, opdelt i lille byte/store byte (2 8-bit registre sat i forlængelse). I programtælleren ligger adressen på den næste instruktion, der skal udføres. Den styrer en hel række interne operationer, antydet helt til venstre, og en opgave kan se således ud:

> Find næste kommando angivet i PC;
> hent den og læg den i instruktionsregistret;
> Dekod kommandoen v.hj.a. instruktioner i en ROM;
> Udfør kommandoen.

Disse trin tager forskellig tid, men det antal taktslag operationerne varer, ses i skemaet over operationskoder. Hver gang en byte er hentet til behandling, fremskrives PC med 1. Derved vil PC altid pege på det næste byte, der skal behandles.
Foruden .a-reg. findes 3 registre mere: .x-reg., .y-reg. og SP. .x og .y kaldes indexregistre. De er begge 8 bit og kan rumme data, men anvendes hovedsagelig som tællere under tal- eller tekstbehandling. Der findes simple kommandoer, der forhøjer(incrementerer) eller nedskriver(decrementerer) indholdet i .x og .y med 1. Indholdet i disse registre kan lægges til en adresse for på denne måde at fremkalde en forskydning. Kommandoerne til .x og .y er ikke helt symmetriske, men det kan ses af tabellen.
SP indeholder en pegepind (Stack Pointer), der peger mod toppen af stakken @:næste ledige plads i stakken af gemte oplysninger. Vi kan fra et program lægge data "på stakken" og hente dem igen. Stakken fylder 1 side i lageret fra $01ff og nedad til $0100. Den virker efter princippet "først ind - sidst ud". Sammenlign stakken med en stabel tallerkener på disken i et cafeteria. Der er en brønd under disken med en fjeder i bunden. Tallerkener stables i brønden og på den måde bliver den først anbragte tallerken den sidst brugte; - den sidst aflagte ligger øverst. Det først aflagte element på stakken lægges på adresse $01ff og SP flyttes 1 plads ned. Næste element skubbes ind på adresse $01fe og SP flyttes 1 ned. Når elementerne hentes fra stakken, opskrives SP tilsvarende og den sidste oplysning, der hentes ligger i $01ff.
Via stakken kan processoren holde styr på spring til og fra subrutiner, idet adressen på næste ordre i hovedprogrammet gemmes.
Vores brug af stakken kan ske med kommandoen "push" og "pull", således at "push" skyder indholdet i .a over på stakken; "pull" henter øverste byte fra stakken til .a. Det er også muligt at flytte indholdet andre steder hen, men transport fra .a til stakken vil vi ofte møde.

Så har vi været gennem talsystemer, datamatens hukommelse og mikroprocessorens opbygning og indre virkemåde. - Nu må vi igang med at programmere. Vi starter lidt forsigtigt. men det er klogt at lave alle opgaver med.

Den første maskinkode.

Dette eksempel skal antyde de overvejelser, du må gøre dig inden du kan få datamaten til at løse en opgave.
Den simple operation:

> Løs opgaven 25 + 9 ved at addere i maskinkode,

må begynde med en del aftaler. 25 kalder vi operand 1, 9 er operand 2, sum giver vel sig selv. Vi råder over lagercellerne $fb til $ff i datamaten og vi vil bruge dem således:

```
operand 1 lægges i celle $fb,
operand 2 lægges i celle $fc og
sum       lægges i celle $fd.
```

Nu kan vi skrive fremgangsmåden:

> Hent operand 1 fra celle $fb og læg den i .a
> Forbered CPU til addition; læg menteflaget
> Adder indhold fra celle $fc til indhold i .a
> Læg resultatet i celle $fd
> Vend tilbage til COMAL.

Denne opskrift kan du se her i maskinkode:

```
10100101
11111011
00011000
01100101
11111100
10000101
11111101
01100000
```

Før du opgiver; så læs lidt videre. Opskriften ser sikkert ud som volapyk, men det er denne opskrift datamaten reagerer på. Hvis vi oversætter til hex-tal, er det lidt mere "venligt".

```
$a5
$fb
$18
$65
$fc
$85
$fd
$60
```

- men stadig meget uoverskueligt. Derfor er der udviklet en masse nøgleord, alle på 3 bogstaver - mnemonics - der skal støtte tanken i opskriften. De er alle dannet ud fra de engelske kommandoer og her følger de første koder:

```
lda  -  load akkumulator; hent til .a
clc  -  clear carry; læg menteflaget
adc  -  addition with carry; adder med mente
sta  -  store akkumulator; læg indhold fra .a .....
rts  -  return from subrutine; tilbage til COMAL
```

Nu kan vi skrive:

```
                    kommentarer
lda $fb     ; læg indhold fra celle $fb i .a
clc         ; læg menteflaget
adc $fc     ; adder indhold fra $fc til indhold i .a
sta $fd     ; læg resultat i celle $fd
rts         ; retur til COMAL
```

- ikke sandt. Oversigten er forøget. Denne skrivemåde kaldes mnemonic kode. Den anvendes i programmeringssproget Assembler.
I programmers ref.guide side 416-417 findes et skema med alle koder. Det findes også på næste side. Se i skemaet for at orientere dig om opbygningen. - Idet cellerne ligger på side 0 (zero-page), skal du nu forsøge at kode ovenstående opgave i hånden. Det er en stor hjælp at lave et skema, som her antydet:

label	hex-tal kommando 1 2 3	mnemonic op-kode operand	bemærkning
	a5 fb	lda $fb	; indh. fra $fb i .a
		clc	
		adc $fc	
		sta $fd	
		rts	

Når du har assembleret opgaven kommer næste problem. Hvordan skal dette program lægges i datamatens hukommelse ? Det kan klares med følgende COMAL-program, idet vi har vedtaget at lægge maskinkoderne fra celle $c000 og frem. Selve programmet er kædet sammen med 2 inputsætninger, der tager sig af de indtastede tal og en peeksætning, der skal vise resultatat på skærmen. COMAL-programmet kalder maskinkoden med SYS($c000). Programmet findes udlistet herunder, men det findes også på disketten under navnet "ma.addition1".

```
// save "@0:ma.addition1"
// delete "ma.addition1"
// * demonstrerer addition af 2 byte
// * og indlægning af maskinkode.
//
adr:=$c000
LOOP
  READ byte
  POKE adr,byte
  adr:+1
  EXIT WHEN EOD
ENDLOOP
PAGE
INPUT "Indtast 1. tal ": tal1
INPUT "Indtast 2. tal ": tal2
POKE $fb,tal1
POKE $fc,tal2
SYS($c000)
PRINT PEEK($fd)
PRINT AT 7,2: "Færdig !"
//
DATA $a5,$fb //  lda $fb
DATA $18 //      clc
DATA $65,$fc //  adc $fc
DATA $85,$fd //  sta $fd
DATA $60 //      rts
```

INSTRUCTIONS		Immediate			Absolute			Zero Page			Accum.			Implied			(Ind.) X			(Ind.) Y			Z, Page, X			Abs. X			Abs. Y			Relative			Indirect			Z, Page, Y		
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3									
AND	AΛM → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3									
ASL	C ← [7 0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3												
BCC	BRANCH ON C = 0 (2)																															90	2	2						
BCS	BRANCH ON C = 1 (2)																															B0	2	2						
BEO	BRANCH ON Z = 1 (2)																															F0	2	2						
BIT	AΛM				2C	4	3	24	3	2																														
BMI	BRANCH ON N = 1 (2)																															30	2	2						
BNE	BRANCH ON Z = 0 (2)																															D0	2	2						
BPL	BRANCH ON N = 0 (2)																															10	2	2						
BRK	(See Fig. 1)													00	7	1																								
BVC	BRANCH ON V = 0 (2)																															50	2	2						
BVS	BRANCH ON V = 1 (2)																															70	2	2						
CLC	0 → C													18	2	1																								
CLD	0 → D													D8	2	1																								
CLI	0 → I													58	2	1																								
CLV	0 → V													B8	2	1																								
CMP	A − M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3									
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																														
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																														
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3												
DEX	X − 1 → X													CA	2	1																								
DEY	Y − 1 → Y													B8	2	1																								
EOR	A∀M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3									
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3												
INX	X + 1 → X													E8	2	1																								
INY	Y + 1 → Y													C8	2	1																								
JMP	JUMP TO NEW LOC.				4C	3	3																												6C	5	3			
JSR	(See Fig. 2) JUMP SUB				20	6	3																																	
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3									
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3												
LSR	0 → [7 0] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3												
NOP	NO OPERATION													EA	2	1																								
ORA	AVM → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3									
PHA	A → M_S S − 1 → S													48	3	1																								
PHP	P → M_S S − 1 → S													08	3	1																								
PLA	S + 1 → S M_S → A													68	4	1																								
PLP	S + 1 → S M_S → P													28	4	1																								
ROL	↳[7 0] ← [C] ↲				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3												
ROR	↳[C] → [7 0] ↲				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3												
RTI	(See Fig. 1) RTRN INT													40	6	1																								
RTS	(See Fig. 2) RTRN SUB													60	6	1																								
SBC	A − M − C̄ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3									
SEC	1 → C													38	2	1																								
SED	1 → D													F8	2	1																								
SEI	1 → I													78	2	1																								
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3									
STX	X → M				8E	4	3	86	3	2																												96	4	2
STY	Y → M				8C	4	3	84	3	2													94	4	2															
TAX	A → X													AA	2	1																								
TAY	A → Y													A8	2	1																								
TSX	S → X													BA	2	1																								
TXA	X → A													8A	2	1																								
TXS	X → S													9A	2	1																								
TYA	Y → A													98	2	1																								

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED.
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE.
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE.
(3) CARRY NOT = BORROW.
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT.

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
M_S MEMORY PER STACK POINTER

\+ ADD
− SUBTRACT
Λ AND
V OR
∀ EXCLUSIVE OR

✓ MODIFIED
— NOT MODIFIED
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
N NO. CYCLES
\# NO. BYTES

Dine forberedelser går bl.a. ud på at bestemme, hvor programmet skal ligge i hukommelsen og hvor operanderne skal placeres. Gør meget ud af disse overvejelser. Tegn evt. den del af hukommelsen, hvor leddene skal lagres.

Hukommelse celle:		
$fb	operand 1	tal1 pokes i $fb
$fc	operand 2	tal2 pokes i $fc
$fd	sum	sum peekes fra $fd

Afprøv programmet med forskellige værdier. Hvor store tal kan du benytte ? Kan 0 bruges ? Kan du bruge negative tal ?

I eksemplet har vi benyttet zero-page til operanderne, men der er pladserne hurtigt brugt op. Hvad gør vi, hvis tallene ikke kan være i en celle ? - Så må hvert tal fylde 2 celler (Vi nøjes med at behandle heltal) - og vi vedtager, at heltal i COMAL skal angives som store byte/lille byte. Adresser skal derimod skrives lille byte/store byte, når programmerne assembleres.
Den adresseringsmåde der her skal benyttes, findes i søjlen "absolute".

Nu vil vi lægge 3 tal sammen. De må godt være større end 256.
Følgende plan opstilles:

Program skal starte i $c000
3 tal indtastes, deles i store byte/lille byte
og pokes på plads i lageret
programmet startes og
oplysningerne fra sum vises
på skærmen.

$c100	store byte sum
$c101	lille byte sum
$c102	store byte tal(1)
$c103	lille byte tal(1)
$c104	store byte tal(2)
$c105	lille byte tal(2)
$c106	store byte tal(3)
$c107	lille byte tal(3)

husker du:

(tal(a) DIV 256) og (tal(a) MOD 256) ?

Programforslag findes på disketten under navn "ma.addition2". List programmet ud på printer og forklar din makker, hvordan tallene pokes på plads i hukommelsen.
I assemblerprogrammet ses et par nye udtryk:

```
lda #$00 betyder: læg værdien $00 i .a ; umiddelbar adress.
sta $c100 skrives med lille byte først ; absolut adressering.
```

Opgaven må være:

Nulstil sum, læg menteflag
Læg lille byte tal(1) i .a
adder med mente lille byte tal(2)
gem resultatet i lille byte sum,
læg store byte tal(1) i .a
adder med mente store byte tal(2)
gem resultatet i store byte sum
slet mente
læg lille byte sum i .a
adder med mente lille byte tal(3)
gem resultatet i lille byte sum

```
læg store byte sum i .a
adder med mente store byte tal(3)
gem resultatet i store byte sum
hop tilbage til COMAL.
```

Prøv, om du kan følge tanken ved at gennemgå sammenlægningen på hukommelsesplanen, angivet på forrige side.

Det er også muligt at subtrahere to tal. Vi springer lige ud i subtraktion af tal, større end 256 og laver denne plan over hukommelsen:

```
$c100  forskel store byte
$c101  forskel lille byte
$c102  operand 1 store byte
$c103     -    1 lille byte  1.operand=minuend
$c104  operand 2 store byte
$c105     -    2 lille byte  2.operand=subtrahend
```

For at kunne subtrahere anvendes instruktionerne

```
sbc - subtract with carry
sec - set carry; sæt menteflag
```

Er menten stadig sat, har der ikke været lånt. - Programmet ser sådan ud:

```
sec         ; sæt menten
lda $c103   ; lille byte minuend til .a
sbc $c105   ; subtraher indholdet i $c105 fra .a
sta $c101   ; gem resultatet i lille byte forskel
lda $c102   ; store byte minuend i .a
sbc $c104   ; sub.store byte op 2 fra op 1
sta $c100   ; gem resultatet i store byte forskel
rts         ; hop tilbage til COMAL

udskriv resultatet på skærmen.
```

Øvelse: Forsøg udfra eksemplet "ma.addition2" om du kan skrive et program, der finder forskellen mellem 2 tal. Under indtastningen bør undersøges, om minuenden>subtrahenden. - Der ligger et forslag til løsning af opgaven på disketten under navn "ma.subtraktion1".

Indtil nu har vi skrevet et par småopgaver som datamaten skulle løse. I afsnittet om 6510-processorens indre bygning blev omtalt et operationsregister. Vi gik let hen over det dengang, men bør uddybe funktionen lidt mere nu.

I vore prograeksempler angiver vi programstart SYS($c000) og så klarer processoren selv resten. Hvordan kan den vide om næste byte, der hentes ind er en adresse, en ops-ordre eller en værdi ?

PC registret er programtæller. I vores eksempel stilles tælleren på $c000, så hentes indholdet af denne adresse til ops-instr.reg.; PC går 1 skridt frem og peger på næste byte. 1. byte dekodes og processoren kan af koden aflæse, om næste byte er et datum, en adresse eller en ny instruktion. Ops-ordren udføres v.hj.a. .a, .x, .y og ALU'en.
Hvis næste byte er en adresse, er der mindst 2 muligheder: a) den kan ligge på side 0, b) den kan ligge på en anden side. Derfor må instruktionerne have hver sin kode og derfor er det vigtigt at forstå forskellen mellem de forskellige måder at adressere processoren på.

En ting, der må understreges: I maskinkode findes ingen fejlmeldinger; der opfanges ingen fejl og datamaten vender ikke tilbage til COMAL af sig selv. Er der fejl i programmet, går mikroprocessoren trøstigt videre; - måske ind i en endeløs sløjfe eller en opgave, der ingen fornuft har. Det kan kun stoppes ved at slukke for hele opstillingen.
Når vi skriver vore små rutiner, bør de altid slutte med rts - return from subrutine - idet COMAL opfatter maskinkodedelen som en underrutine.

Brug af index-register.

Da du skulle lægge 3 tal sammen, var det nødvendigt at springe fra første sammenlægning til sum og fra anden sammenlægning til sum, idet der ellers evt. ville blive kludder med menten. Hvis du skulle lægge 10 tal sammen, ville det blive uoverskueligt og du skulle skrive de samme kommandoer mange gange. Samtidig kræver programmet megen lagerplads, så der må være en smartere opskrift.
Indtil dette tidspunkt har vi brugt den absolutte adresse: Lda $c102, lda $c104...... Metoden kaldes absolut adressering. - I det næste forslag opgives kun startadressen og ved hjælp af en tæller kommer vi frem til næste celle. Denne metode kaldes indexeret adressering. Tælleren kan ligge enten i .x eller i .y.
Den ønskede adresse findes med x-indexeret adressering ved at indholdet i .x lægges til startadressen. Under programafviklingen kan indholdet i .x ændres. Lad os antage, at vi skal lægge 5 tal sammen og at summen ikke overstiger 255. $c100 skal rumme værdien "sum" og antal led ligge i $c101. Leddene ligger i celle $c102, $c103,.......
Tegn hukommelsen med cellerne belagt og forsøg at følge med i denne opskrift:

```
       Addition af flere tal (sum=<255)

       ldx #$00       ; pointer til data
       txa            ; transfer .x til .a , altså $00
       clc            ; læg mente
hop    adc $c102,x    ; addition
       inx            ; næste byte
       cpx $c101      ; er tæller=antal
       bne hop        ; hvis nej - spring til hop
       sta $c100      ; hvis ja - gem i sum
       rts            ; retur til COMAL
```

Her var nogle nye op-koder:

```
       ldx  -  load indexreg. x
      #$00  -  umiddelbart med værdien $00
       txa  -  transfer .x til .a, kopier indh. i .x til .a
       inx  -  increment indexreg. x, forhøj indhold i .x med 1
               er indholdet allerede $ff, sker overløb til $00
               og zero-flag sættes
       cpx  -  compare .x med celle......., sammenlign indhold
               i .x med indhold i celle......
               Er resultatet 0, sættes zero-flag
       bne  -  branch on not equal zero, hop hvis testen ikke
               giver 0 (zero-flag ikke sat). Det antal bytes
               operanden angiver for spring er i tokomplement.
       hop  -  er et mærke, hvortil springet skal ske.
```

Når operanden "hop" er indlæst efter bne, peger PC på "sta" - tæller du bagud til mærket "hop", altså til ops-ordre adc, når du til -9 byte. PC skal altså stilles 9 byte tilbage. Denne off-set

skal udtrykkes i tokomplement:

```
9 er binært       %00001001
enkomplement      %11110110
            +             1
tokomplement      %11110111
omsat i hex       $ f    7
off-set er altså    $f7
```

Prøv at lave det samme skema, som angivet tidligere, assembler programmet i hånden, indskriv det i et COMAL-program og få det til at køre. Se evt. "ma.addition3".

16-bit addition af flere led.

Hvis sum skal være større end 255, må de enkelte led også fylde mere end 1 byte. Hvert led må altså fylde 2 byte. Antal led skal stå i $c100, $c101 og $c102 skal rumme sum, fra $c103 ligger de enkelte tal i store byte/lille byte.
Lav en tegning af hukommelsen; - den skal rumme sum + tæller + 5 tal. Følg dernæst med i disse overvejelser:

```
ldx #$00      ; læg værdien $00 i .x
stx $c101     ; nulstil $c101 - i start er sum 0
stx $c102     ; det samme gælder her
```

Da hvert tal fylder 2 byte, må indholdet i $c100 fordobles for at tælleren kan nå ud til yderste led. Så må ALU'en forberedes til sammenlægning; - menten slettes.

```
clc           ; læg mente-flag
lda $c100     ; hent indhold i $c100 til .a
adc $c100     ; adder med mente indhold i $c100
sta $c100     ; gem resultatet i $c100 - tæller fordoblet
```

Her følger den endelige sammenlægning i en sløjfe. Slet mente og begynd fra oven med tal(5) lille byte.

```
    tax           ; kopier indhold i .a til .x
hop clc           ; slet mente
    dex           ; nedskriv indhold i .x med 1
    lda $c102     ; læg lille byte sum i .a
    adc $c103,x   ; adder lille byte tal(5)
    sta $c102     ; gem resultatet i lille byte sum
    dex           ; gør indh. i .x 1 mindre
    lda $c101     ; læg store byte sum i .a
    adc $c103,x   ; adder store byte tal(5)
    sta $c101     ; gem resultatet i store byte sum
    cpx #$00      ; sammenlign .x med $00 - er jeg færdig ?
    bne hop       ; hvis nej - så hop
    rts           ; hvis ja - så returner
```

Når hopordren skal udføres, peger PC på rts og der skal springes 25 bytes bagud. (Tæl efter om det passer). Springet skal angives i tokomplement.

```
25 er binært      %00011001
enkomplement      %11100110
           +              1
tokomplement      %11100111
     hex             e   7
off-set er            $e7
```

Her var der også nye ops-ordrer:

```
        stx   -   store index i memory; gem indh. .x i celle.....
        tax   -   transfer .a to .x; kopier indh. .a til .x
        dex   -   decrementer .x; nedskriv værdi i .x med 1
```

Kik på din tegning af hukommelsen og gennemgå løkken, så du er fortrolig med dens virkemåde. Assembler programmet i hånden, indskriv det i et COMAL-program og få det til at virke.
Model ligger på disketten som "ma.addition4".

Øvelse: Hvad sker der, når disse ops-ordrer udføres ?

```
  a) lda #65            b) ldx #$00           c) lda #65
     sta $0401  ?          txa                   ldy #$05
                           sta $c103  ?          sta $0401,y  ?
```

d) Hvad er forskellen på lda og sta ?

e) Hvad sker i .y ved ops-ordre iny ?

f) Hvad betyder rts ? 6) Hvad er store byte af decimal 1030 ?

Fremstilling af subrutiner.

Skal en operation udføres flere gange, kan det være hensigtsmæssigt at skrive den som en underrutine (subrutine), der kaldes hver gang der er brug for den. Parametre, der skal overføres kan lægges i .x og .y før rutinen kaldes. Som eksempel vil vi gennemføre addition af 3 tal.
De sædvanlige aftaler skal laves. Først planen for cellernes belægning:

Hukommelse

```
$c100 store byte sum
$c101 lille byte sum
$c102 store byte tal(1)
$c103 lille byte tal(1)
$c104 store byte tal(2)
$c105 lille byte tal(2)
$c106 store byte tal(3)
$s107 lille byte tal(3)
```

De to byte, der skal adderes til sum, lægges i .x og .y med lille byte i .x, store byte i .y. Sum lægges tilbage til cellerne $c100 og $c101 og derefter springes tilbage til hovedprogrammet. Denne subrutine kan lægges hvor du ønsker det i forhold til hovedprogrammet, idet den kaldes med jsr adresse og sluttes med rts.

Vi vil lægge den fra adresse $c030, men det giver lidt besvær at få den læst ind. Se eksemplet på disketten "ma.addition5".
Når subrutinen kaldes med "jsr adresse", må processoren vide, hvor den skal fortsætte i hovedprogrammet efter udførelsen. Derfor gemmes næste adresse på stakken (stack register). Så lægges adressen fra jsr i PC og afviklingen fortsætter fra denne adresse.
Slutmærke på subrutinen er ops-ordre rts, der bevirker spring tilbage til hovedprogrammet - og samtidig hentes adressen fra stakken. Den lægges i PC og programmet fortsætter.

```
Subrutinen lægges fra $c030
clc       ; læg mente
txa       ; kopier lille byte til .a
adc $c101 ; læg hertil sum
sta $c101 ; og læg ny sum tilbage
tya       ; kopier store byte til .a
adc $c100 ; læg hertil sum
sta $c100 ; læg ny sum tilbage
rts       ; return fra subrutine
```

S-mon - Et monitorprogram.

Denne monitorvejledning er en udvalgt del af den komplette vejledning til SMON. Den kan bl.a. findes i 64'er Sonderheft 8/85.
Programmet findes med forskellige startadresser, så det kan placeres i C-64 sammen med det maskinprogram, der skal undersøges.
En monitor er et stykke værktøj hvormed du kan udskrive en del af lageret til skærm eller printer, du kan ændre og save den nye udgave, disassemblere, loade et maskinkodeprogram og starte det; - forskyde programdele, omregne mellem binær- og hexnotation, - søge efter bestemte kombinationer. Alle indtastninger skal være i hextal. Når der i vejledningen skrives (start), betyder det den nøjagtige startadresse; - (slut) betyder derimod den første frie adresse efter det valgte udsnit. Der tages ikke hensyn til mellemrum. Din første brug af monitoren vil nok være at disassemblere færdige maskinkoder for at lære, hvordan de forskellige problemer løses. I vejledningen arbejdes med udgaven, lagt fra $c000, men de samme kommandoer gælder for udgave $9000.

Start:

```
Basic <R>
LOAD "SMON$CO 49152",8,1 <R>
NEW <R>
SYS 49152 <R>
```

Herefter ses en statuslinje efterfulgt af et punktum og den blinkende markør:

PC	SR	AC	XR	YR	SP	NV-BDIZC
C00B	B0	C2	00	00	F6	10110000

.█

Assemblere
.A(start) <R>   eks.: .A 4000
assembleringen begynder ved den opgivne adresse. Nu har du mulighed for at indskrive kildetekst. Skriver du fejl, springer markøren tilbage til linjens start for nyt forsøg. Ved lovlig indtastet linje, opgives hexkode og den assemblerede linje. Det er muligt at rette ved overskrivning af en linje. Hvis springadresse endnu ikke kendes, bruges label. Her bruges M og et tocifret hextal for springmål. Når linjer skrives, hvortil et spring skal ske, sætter man den samme label foran ops og operand. Når du er færdig med assembleringen, skrives F <R> og indtastningerne listes på skærmen.

.S"navn",(start)(slut) <R>
gemmer et program fra startadresse til slutadresse-1 under "navn" på disketten. Programmet er indstillet til enhed nr 08.
I01 <R> omstiller til kassetteoptager.

.L"navn" <R>
henter et program fra den opgivne enhed og lægger det på det rigtige sted i lageret. NB. se bemærkningen.

I/O SET
I01 <R> omstiller datamaten til kassettebrug.
I08 <R> omstiller datamaten til floppy.

PO SET
Ved alle output kan printer anvendes. Kommandoen indgives med skiftenøgle + kommandotast. Herefter udprintes resultatet samtidig

med at det vises på skærmen.

Disassemblering
.D(start)(slut) <R>
disassemblerer lagercellernes indhold fra startadresse til slutadresse-1. Hvis slutadresse ikke indtastes, vises kun 1 linje på skærmen; - tryk på mellemrum fremkalder næste linje. Trykker du <return>, -løber linjerne over skærmen, indtil du trykker på en tast, - eller slutadr. nås. RUN/STOP bringer dig til indtastmode. Vil du ændre, kan du overskrive en linje. Pas på at alle cellerne fyldes ud. En tom plads kan fyldes op med NOP.

Start et program:
.G(start) <R>
starter et maskinkodeprogram, der begynder ved (start). Programmet må slutte med BRK, så der kan springes tilbage til SMON. I modsat fald, må SMON startes påny med SYS 49152 <R>.

Memory Dump
.M(start)(slut) <R>
viser hexkoder i lageret og samtidig de ASCII-koder, der kan oversættes. Her kan slutadresse også undlades og så sker styringen som under disassemblering.
.M 4000 viser lagercellerne 4000 til 4007. Der kan fortsættes med space eller return. Der kan overskrives i lageret, men ikke i oversættelsen.

Udgang
.X <R>
springer tilbage til BASIC. Alle pointere bliver urørte. Du kan altså fortsætte programmet og springe retur til SMON senere med SYS 49152.

Omregning
#decimaltal <R>
omregner til hex-tal
$hexadecimaltal <R>
omregner til decimaltal
%binærtal <R>
omregner til hex- og decimaltal. Skal indtastes som 8 tegn.
?2340+156d <R>
giver sum i hex. - Kan også vise forskel

.O(start)(slut)hex <R>
belægger området fra start til slut med hexværdi.
eks.: .O 3000 4000 ff <R>
fylder alle adresser fra $3000 til $3fff med $ff.
.W(startgl)(slutgl)(startny) <R>
eks.: .W 4000 4024 6000 <R>
forskyder lagerområdet fra startgl til til slutgl hen til startny og videre frem. Adresser omregnes ikke.
.V(startgl)(slutgl)(startny)(start)(slut) <R>
eks.: .V 4000 4024 6000 4012 4024 <R>
omregner alle absolutte adresser i området fra start til slut, som befinder sig mellem startgl og slutgl til nye adresser i forhold til startny.
Er det indviklet ? Tænk på at de første 3 adresser svarer til oplysningerne i .W. - Hertil føjes kun de to adresser hvor imellem omregningerne skal udføres.
.C(startgl)(slutgl)(startny)(startfor)(slutfor) <R>
forskyder et program fra startgl til slutgl hen til startny og omregner adresserne mellem startfor og slutfor.

.B(start)(slut) <R>
omdanner maskinkodeprogram fra start til slut-1 til basicdata-linjer. Der begyndes med linje 32000 og de lægges i basicområdet. Hvis der findes et ladeprogram med lavere linjenumre, kan du starte direkte.
.K(start)(slut) <R>
udskriver ASCII-tegn der findes i lageret fra start til slut-1. Som ved disassemblering kan anvendes space og return for at komme gennem et program. Det er muligt at overskrive ASCII-tegnene i koden.
.F(hex-værdi),(start)(slut) <R>
søger efter enkelte - eller flere byte i området start til slut-1. Flere tegn må skrives med mellemrum mellem de enkelte byte.
.FA(adresse),(start)(slut) <R>
leder efter alle ops, der har en bestemt adresse som operand. Her kan godt bruges wildcards, altså *
.FR(adresse),(start)(slut) <R>
leder efter relative spring til adresse i området start til slut-1.
.FT(start)(slut) <R>
leder efter tabel i det angivne område. Alt hvad der ikke kan dis-assembleres opfattes som tabeller.
.F2(start)(slut) <R>
leder efter alle ops med zero-page som operand.
.FI(operand),(start)(slut) <R>
leder efter alle ops med umiddelbar adressering.

Eksempel på brug af SMON

Der foreligger følgende lille assemblerprogram:

```
        lda #$93
        jsr $ffd2
        ldy #$00
  m2    lda m1,y
        jsr $ffd2
        iny
        cpy #$12
        bne m2
        brk
  m1    .byte $2a
        .byte $2a
        .byte $20
        .byte $48
        .byte $45
        .byte $4a
        .byte $20
        .byte $4d
        .byte $45
        .byte $44
        .byte $20
        .byte $44
        .byte $49
        .byte $47
        .byte $21
        .byte $20
        .byte $2a
        .byte $2a
```

Dette program skal nu assembleres og lægges fra $4000. Du skal bruge SMON og operanderne indtastes her uden $. Når du er færdig, tastes <F> og programmet listes.

.G 4000 <R> starter programmet.
.S"navn",start slut <R> og programmet er gemt på disk.
.M 4000 4020 <R> viser programmet i hukommelsen.
.D 4000 4013 <R> disassemblerer programmet - tabellen.

Programmet ligger fra 4000 til 4024. Vi vil gerne have skriften til at starte fra 4. linje 6. plads, det ser i assembler sådan ud:

```
ldy #$04  ; linje
sty $d6
ldx #$06  ; plads
stx $d3
jsr $e56c
```

Denne tilføjelse fylder 11 pladser, eller i hex $0b pladser. Den skal skydes ind mellem clear screen og 4005. Alt fra 4005 skal altså skubbes $0b pladser op i lageret. Det sker sådan:
?4005+0b <R> giver 4010. Programmet efter indskud skal dermed flyttes til 4010.
.C 4005 4025 4010 4005 4012 <R>
Derefter kan indskudet skrives.
.A 4005 <R>
Indføj programstumpen. Flyt markøren ned under sidste linje og tast return.
.M 4000 4030 <R> Ligger fornyelsen korrekt ?
.G 4000 <R> Fungerer det efter ønske ?
.S"nytnavn" 4000 4030 <R>
.O 4000 4030 ff <R> Hele området belægges med ff.
.L"nytnavn" <R> Programmet lægges i lageret.
.M 4000 4030 <R> Kontrol af dette.
Bemærkning:
Nu skal programmet lægges et andet sted i hukommelsen eks. fra 5000.
.L"nytnavn" 5000 <R>, men adresser ændres ikke. Programmet kan ikke afvikles, men det er metoden, hvis et program med autostart, fra f.eks. $0120 skal disassembleres. Hvis du indtaster .L"eksempel" 4120 <R>, lægges programmet herfra og ikke fra originaladressen $0120. Det er nu muligt at undersøge programmet.

```
,4000  A9 93     LDA #93
,4002  20 D2 FF  JSR FFD2
,4005  A0 00     LDY #00
,4007  B9 13 40  LDA 4013,Y
,400A  20 D2 FF  JSR FFD2
,400D  C8        INY
,400E  C0 12     CPY #12
,4010  D0 F5     BNE 4007
```

```
,4000  A9 93     LDA #93
,4002  20 D2 FF  JSR FFD2
,4005  A0 04     LDY #04
,4007  84 D6     STY  D6
,4009  A2 06     LDX #06
,400B  86 D3     STX  D3
,400D  20 6C E5  JSR E56C
,4010  A0 00     LDY #00
,4012  B9 1E 40  LDA 401E,Y
,4015  20 D2 FF  JSR FFD2
,4018  C8        INY
,4019  C0 12     CPY #12
,401B  D0 F5     BNE 4012
```

```
:4000 A9 93 20 D2 FF A0 04 84   .. —....
:4008 D6 A2 06 86 D3 20 6C E5   ×...♥ ..
:4010 A0 00 B9 1E 40 20 D2 FF   ....@ —.
:4018 C8 C0 12 D0 F5 00 2A 2A   ⊢—.⅂..**
:4020 20 48 45 4A 20 4D 45 44    HEJ MED
:4028 20 44 49 47 21 20 2A 2A    DIG! **
:4030 00 FF 00 FF 00 FF 00 FF   ........
:4038 00 FF 00 FF 00 FF 00 FF   ........
```

Tidsbestemmelse.

Her skal undersøges, hvilke muligheder datamaten har for at bestemme et tidsrum. Den første ide går ud på at lade den tælle i COMAL. Indtast dette program og juster på tællingen, indtil der går nøjagtig 60 sekunder, før "Færdig !" ses på skærmen. Det er bedst at kontrollere med stopur.

```
page
print at 5,5: "Tast når tælling skal begynde."
while key$=chr$(0) do null
for a:=1 to 60*875 do null // skal måske tilpasses
print at 9,5: "Færdig !"
```

I dette eksempel, skal datamaten åbenbart tælle til 875, før der er gået 1 sek.
I systempakken findes en funktion TIME, der måske kan bruges også.
Modificer programmet, så det ser således ud:

```
USE system
page
print at 5,5: "Tast........."
while key$=chr$(0) do null
time 0
for a:=1 to 60*875 do null
print at 7,5: time
print at 9,5: "Færdig !"
```

Hvor mange skridt er time gået frem på 60 sek?
Datamatens indre tæller går frem i trin, vi kalder "øjeblikke".
Passer det med dine målinger, at datamaten bruger 60 øjeblikke til 1 sek? - Disse øjeblikke kaldes også for jiffies. Hvad vil der ske med vores tidsmåling, hvis datamaten skal udføre en opgave på skærmen samtidig med at den tæller? Hvis der nu bare skal skrives et par linjer; har det mon nogen effekt - det hele går jo hurtigt? Modificer programmet, så der efter time 0 skrives en 3-4 linjer tekst på skærmen, inden tælleren starter. Har det nogen effekt?
Er det kun en grov tidsmåling, der skal foretages, kan det være godt nok at lade datamaten tælle, men en mere nøjagtig tidsmåling vil anvende det indbyggede ur, der fra COMAL kaldes med TIME. Tællelinjen i eksemplet ovenover kunne udformes således:

```
while time<3600 do null
```

Prøv om det giver bedre resultat. Se evt. eksemplerne på disketten "tællerforsøg 1-5".

Der er også mulighed for direkte at arbejde med de indbyggede tællere, og så må vi tage fat på at undersøge

Tællerne i datamaten.
På vores prøveplade findes en kontakt med betegnelsen CNT2. CNT er en forkortelse for tæller. Der findes 2, men CNT1 er ofte optaget af internt arbejde, så derfor vil vi koncentrere os om CNT2.
Tælleren kan enten tælle impulser der kommer udefra eller clock-impulser fra datamatens egen klokke.

Når datamaten tændes, ligger der +5 V på CNT2, og her tælles positive impulser, d.v.s. tælleren går 1 skridt frem hver gang en firkantimpuls går fra 0 V til +5 V. Før tælleren aktiveres, må det fastlægges hvordan, den skal arbejde. Disse oplysninger lægges i kontrolregisteret.

Tælleren kan sammensættes af 4 stk. 8-bit tælleregistre, der kan lægges i forlængelse af hinanden, men først en oversigt over datamatens tællere:

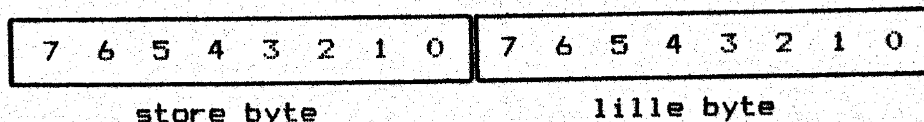
```
$dd04  timer/tæller A lille byte
$dd05  timer/tæller A store byte
$dd06  timer/tæller B lille byte
$dd07  timer/tæller B store byte
$dd0e  kontrolregister A
$dd0f  kontrolregister B
```

I timerregistrene findes oplysninger om antal opfangede signaler.
I kontrolregistrene kan lægges følgende oplysninger:

```
Kontrolregister A
Bit nr.    Funktion
  0        0 stop timer A
           1 start timer A
  1        0 output på PB6 nej
           1 output på PB6 ja
  2        0 hvis output på PB6: impuls
           1 hvis output på PB6: flip/flop
  3        0 kontinuert tælling
           1 1 gennemløb, one shot
  4        0 reset til anden værdi, se pr.g.side 429
           1 reset til $ff i begge byte
  5        0 timer A - computer clock
           1 timer A - CNT2 signaler
  6        0 serielport input
           1 serielport output
  7        0 time of day clock 60 Hz
           1 time of day clock 50 Hz

Kontrolregister B
Bit nr.    Funktion
 4-0       Som ovenfor, bare tæller B
 6-5       fungerer sammen således:
           0   0  tæller B tæller på CPU clock
           0   1  tæller positive impulser på CNT
           1   0  tæller hver gang tæller A har underflow
           1   1  tæller underflow A mens CNT er positiv
  7        0 set alarm fra CPU clock
           1 set alarm fra TOD clock
```

Vi lægger ud med tæller A.
Hertil hører kontrolregister $dd0e, lille byte $dd04 og store byte $dd05.
Store byte og lille byte kan sammenkobles til et 16 bit register, der kan tælle til 2↑16 = 65535.



store byte   lille byte

Store byte rummer de mest betydende bits. Når tælleren er aktiveret og modtager tælleimpulser, er det kun lille byte-registeret, der modtager signaler. Hver gang dette register har talt ned til 0 og begynder forfra, overføres 1 impuls til store byte. Indholdet af $dd05 er altså det antal gange, $dd04 er begyndt forfra.

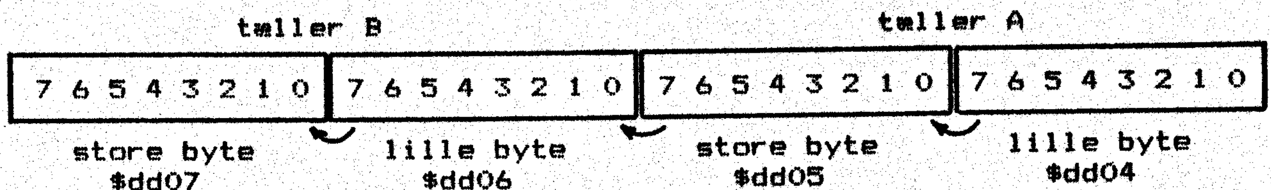
Prøv dette lille eksperiment:
Forbind kontaktpladen med prøvepladen og derfra til userporten.
Indtast derefter dette program:

```
page
poke $dd0e,%00110001 // start, reset, fra CNT2
repeat
  s:=peek($dd05); l:=peek($dd04)
  print at 5,5:using "store byte ###, lille byte ###":s,l
until false
```

Se på linje 2. I kontrolregister A sættes bit 0, 4 og 5 høje, det medfører start af tæller, reset med $ff i registeret og at der tælles pulser ved overgang fra 0 til +5 v på CNT2. Den værdi, der ligger i store og lille byte, skrives ud på skærmen.
Når du starter programmet ses det, at der tælles nedad mod 0 fra 255. Hvis det er nødvendigt for dig, at der tælles opad, må linjen med tildelingerne ændres til s:=255-peek($dd05)........ - lav den selv færdig og kontroller, om ideen fungerer.
Hvordan skal programmet ændres, hvis der skal tælles opad og vi også gerne skal se det totale antal impulser?
Se eksemplerne "tællerøvelse 1-3".

Skal der holdes styr på korte tidsrum med en rimelig nøjagtighed på et par sekunder, eller registreres enkelt-input, kan det klares i COMAL-programmet.
Vokser kravet til nøjagtighed, og skal der evt. afvikles andre opgaver på samme tid, må opgaven løses i assembler. Dette emne behandles senere, hvor vi vil tælle impulser over en lysbro.

Hvis CPU-klokken skal bruges som timer, kan korte intervaller klares med enten tæller A eller tæller B. Længere intervaller kan registreres, når tæller A og tæller B sættes i forlængelse af hinanden.



Her er tæller A og tæller B sat i forlængelse af hinanden. Alle impulser ankommer til tæller A lille byte.
Oplysningerne for at få denne tæller til at fungere, må ligge i kontrolregistrene således:

Starte og stoppe en tæller for CNT2 impulser.
Til kontrolregistret pokes:

```
  Reset og start  %00110001 = $31 // tæller firkant, reset, start
  Stop            %00000000 = $00 // tæller stoppes
```

Tæller A og B koblet sammen, B tæller underflow fra A.

```
  Reg.B           %01010001 = $51 // tæl underflow, reset, start
  Reg.A           %00110001 = $31 // firkant, reset, start
  Stop (A og B)   %00000000 = $00 // tæller stoppes
```

Tidsmåling med CPU clock, hvis 1 tæller er nok.
Til kontrolregistret pokes:

```
  Reset og start  %00010001 = $11 // reset og start
  Stop            %00000000 = $00 // tæller stoppes
```

Hovedprogrammet ses herunder:

```
lda #$00   ; læg 0 i .a
sta $c100 ; nulstil
sta $c101 ; resultatet
ldx $c103 ; første led lille byte
ldy $c102 ; første led store byte
jsr $c030 ; hop til subrutine
ldx $c105 ; andet led lille byte
ldy $c104 ; andet led store byte
jsr $c030 ; hop til subrutine
ldx $c107 ; tredje led lille byte
ldy $c106 ; tredje led store byte
rts       ; tilbage til COMAL
```

Af nye ops-ordre er her brugt:

```
tya  -  transfer indh. .y til .a
jsr  -  jump to subroutine
        skal indeholde startadresse på subrutine
```

Adresseringsmåder.

Det er nok på sin plads at se nærmere på de forskelling adresseringsformer nu. Mikroprocessoren kan på adresseringsmåden aflæse, hvordan en ops-ordre skal behandles og om næste byte er en adresse eller en ny ops-ordre. Hvis vi benytter celler fra side 0, taler vi om zero-page adressering. Her er forløbet kort, idet adressen ikke behøver store byte.

Umiddelbar adressering (immediate) - 2 byte
Her lægges en talværdi i .a, .x eller .y. Ops-koden følges af et tal med et amerikansk nummertegn "#" foran.

```
ldx #%11011000
lda #10
ldy #$ff
cmp #$7f
```

Absolut adressering (absolute) - 3 byte
Her arbejdes med en lagercelle i hukommelsen. Derfor må adressen opgives i 2 byte på formen lille byte/store byte.

```
sta $dd01
cpx $c102
lda $c010
ldy $c1ff
```

Side-0 adressering (zero-page) - 2 byte
Denne adressering ligner den absolutte adressering, men ops-koden følges kun af 1 byte.

```
cpy $fd
inc $ff
ldx $fb
sta $ff
```

Akkumulatoradressering (accumulator) - 1 byte
Her foretages en manipulation inde i .a. Der sker intet i hukommelsen. Derfor kun 1 byte.

```
asl
lsr
rol
ror
```

Implicit adressering (implied) - 1 byte - underforstået adress.
Korte koder, hvor der kun opereres på registre.

```
clc
dex
iny
tax
```

Absolut x-indexeret adressering (abs,x) - 3 byte
Korrekt celle findes ved at MPU'en til absolut adresse lægger indholdet i .x. - Absolut y-indexeret adressering (abs,y) virker ligesådan.

```
lda $c000,y
sta $dd01,x
cmp $dd18,y
adc $c100,x
```

Relativ adressering (relative) - 2 byte
Bruges kun i forbindelse med betingede hopordre. Den aktuelle adresse opgives som tokomplement og muliggør spring 127 byte frem og 128 byte bagud.

```
bne $f6
bcc $08
```

Indirekte adressering (indirect) - 3 byte
Kan kun forekomme med jmp. Lad os antage at $c100 rummer $10, $c101 rummer $c1. Ops-ordre jmp ($c100) bevirker nu at PC springer til $c100 og opfatter indholdet som lille byte; går videre til $c101, hvor $c1 opfattes som store byte i adressen: $c110. I eksemplet vil jmp ($c100) bevirke spring til $c110. Adressen opgives indirekte.
Bemærk, at der også findes et absolut spring: jmp $ffd2.

Endelig får vi brug for indirekte y-indexeret adressering, men denne metode omtales nærmere i afsnitet med skrift på skærmen.

- ø - Ø - ø -

Øvelse:

Er disse assembleringer rigtige ?

```
lda $fb       $a5,$fb          dey            $b8
sta $c110     $8d,$10,$c1      jmp $ffd2      $4c,$d2,$ff
inc $ff       $eb,$ff          rts            $60
lda $c100,x   $bd,$00,$c1      sta $c020,y    $99,$20,$c0
rol           $2a              jmp($c100)     $6c,$00,$c1
beq $fa       $f0,$fa          inx            $e8
```

En simpel monitor.

Når koderne er lagt på plads af COMAL-programmet, ville det så ikke være interessant at se, om de virkelig ligger de rigtige steder ? Det kunne også være spændende at gå på opdagelse forskellige steder i datamatens hukommelse. Begge ønsker kan opfyldes v. hj.a. en monitor.
En monitor er et program, der gør det muligt at vise indholdet i de udvalgte celler på dataskærm eller skrive koderne til printer. Når programmet er startet, skal du indtaste startadresse og slutadresse for det område, du vil undersøge. Adresserne skal indtastes i hex-kode. Derefter går programmet igang med, celle for celle, at udskrive indholdet i hex-kode. Hvis koden er et ASCII-tegn, der kan vises på skærm (et tal eller et bogstav) udskrives dette også; ellers skrives et punktum i oversættelsen. Du kan ikke rette i programmet ved at skrive på skærmen, - kun se indholdet.
Du skal senere se en større monitor, hvormed det er muligt at foretage rettelser. Programmet ligger på disketten under navnet: "se'lager". - Samme program er også list'ed til disketten som en ASCII-fil efter at det er ændret til en stor samlet procedure: "lst.se'lager". Fordelen ved denne fil er, at den kan hægtes til et andet COMAL-program med kommandoen: MERGE "lst.se'lager" <RETURN>. Herefter er proceduren hægtet sammen med det program, der lå i arbejdslageret i forvejen. Ved start af dette program sker intet nyt, men maskinkoden er lagt på plads som sædvanligt. Hvis du herefter skriver direkte til skærmen: se'lager <RETURN>, afvikles monitoren som ovenfor. Prøv det med et eller flere af programmerne, vi allerede har gennemgået.

Enkeltskridt simulator 6510.

Her finder du et stort maskinkodeprogram, der efterligner funktionerne i mikroprocessor 6510. Når programmet er startet, vises indholdet i programtæller (PC), akkumulator (.A), x-register (.X), y-register (.Y), statusregister (SR) og stackpointerens stilling (SP). De enkelte flag i statusregisteret vises også enkeltvis. Du kan ved at følge nedenstående vejledning stille programtæller, lægge bestemte værdier i registre og gå frem gennem et maskinkodeprogram skridt for skridt ved tryk på mellemrumstangent og ved at indtaste de forskellige værdier.
Programmet er formet som en pakke, der kan linkes til det program, du ønsker at undersøge. Se evt. nærmere om maskinkodepakker i COMAL-håndbogen. Programmet findes på disketten som en fil, kaldet: "pkg.simulator", det gøres kendt som "simulator" og procedurekaldet er "simuler".
En maskinkodepakke kan bindes til et COMAL-program med kommandoen LINK "pkg.simulator" <RETURN>. Derefter gøres den kendt ved at du skriver: USE simulator <RETURN> og når du vil bruge programmet, kaldes det med: simuler <RETURN>. Resten af funktionen kan du læse i nedenstående vejledning. Dog bør det nævnes, at du save'er pakken med på disketten, hvis du herefter vil gemme dit program.
Ønsker du at frigøre dit program fra pakken, sker det med kommandoen DISCARD (RETURN>.

Betjeningsvejledning for simulator 6510.

Her præsenteres et program, der kan bruges ved prøvekørsel og evt. fejlfinding i egne maskinkodeprogrammer. Det kan bruges i COMAL, idet objektkoden er placeret i en pakke. Simuler - kan bruges til efterprøvning af egne maskinkodeprogrammer og til træning i assemblerprogrammering. Der startes i COMAL med procedurekald og herefter vises de forskellige registres indhold på skærmen.

PC .A .X .Y SR SP NV-BDIZC

Vi genkender		Flag vi kender:
	PC - programtæller	N - negativ
	.A - akkumulator	V - overflow
	.X - x-register	Z - zero
	.Y - y-register	C - carry (mente)
	SR - statusregister	
	SP - stackpointer	

Undervejs er det muligt at ændre i de enkelte registres indhold. De kaldes med første bogstav i forkortelsen. Det gamle indhold vises under kassen og markøren blinker på 1. tegn. Der kan overskrives med et lovligt hex-tal, hvorefter der slås <R> og den nye værdi indsættes i kassen. Samtidig slettes input-linjen.

Koderne er følgende:

P viser PC gamle indhold. Det kan overskrives med den ønskede adresse og efter <R> overtages denne i displayet. Den tilhørende ordre disassembleres og vises. Flag opdateres.
A viser .A indhold, der også kan ændres ved overskrivning. Efter <R> står den nye værdi i displayet.
X viser .X indhold, analogt med .A
Y viser .Y indhold, analogt med .A
S sender SP til inputlinjen. Også SP kan overskrives.
SR kan ikke ændres uden videre, idet det styres af de forskellige flag. Det er altså flagene, der skal ændres.
N indtastes, hvis negativ-flaget skal omstilles. Funktionen styrer en flip-flop, så hvert nyt tryk på N omstiller flaget. Det samme er tilfældet for de andre flag.
E Her spørges, om der skal rettes i lageret.

Simulatorens vigtigste funktion aktiveres med mellemrumstangenten, idet den maskinkodeordre som PC peger på disassembleres og vises på skærmen samtidig med at ordren udføres.
Det er ikke mikroprocessoren der udfører opgaven, men maskinkode-programmet der simulerer forløbet. Registrene og de aktuelle flag opdateres korrekt.
Ønsker man ikke at udføre en ordre, men lade PC tælle videre, kan det gøres med "markør ned". Hvis der under simuleringen forekommer ordrer, der ændrer på lageret, som f.eks. STA eller INC, kan dette få følger for afviklingen. Der kan gribes ind ved at taste "E". Så vises linjen "Ægte simulering? J". Tastes <R>, udføres den pågældende ordre, Tastes "N" <R>, overspringes den.
M kan vise en lagerplads. Hvis E forud er aktiveret, kan en lagercelle overskrives. M 0002 viser lagercelle 0002's indhold, der evt. kan overskrives.
Ved indexeret, indirekte indexeret og indexeret indirekte adressering, vises i < > den aktuelle adresse samt dennes indhold.
Mellemrum+C= giver hurtig simulering. Programmet forlades med RUN/STOP eller *.

Nu kan du forsøge dig frem med dette program. Det bliver særligt brugbart, når vi skal prøvekøre lidt større maskinkodeprogrammer; og det giver mest indsigt, hvis du har det udlistede program ved siden af dig. Du kan lære meget ved at følge udlistningen. - Tænker du på andre ting, mens du bruger programmet, er indlæringseffekten meget ringe. Det er nemlig ikke et stykke legetøj, eller et spil !

Færdige subrutiner.

Vi har allerede lavet en subrutine selv. På side 29 ligger en rutine, der kan lægge 2 16-bit tal sammen. Mange opgaver rummer rutiner, som skal gentages hyppigt. Derfor findes der, dels i COMAL, dels i Kernal, færdige rutiner, vi kan benytte.

Skal vi hente et tegn fra tastaturet, kan vi benytte:

```
   getin  $ffe4    Rutinen henter en karakter fra keyboard-
                   bufferen og returnerer ASCII-værdien.
                   Hvis bufferen er tom, returneres "0"
                   Værdien findes i .a.
           eks.    vent  jsr getin
                         cmp #$00
                         beq vent
```

Udskrift af et tegn på skærmen klares med:

```
   chrout $ffd2    Tegnet skal ligge i .a inden rutinen kaldes.
           eks.          lda #"A"
                         jsr chrout
```

Markørstyring kan ske ved at styre markørens næste plads eller ved at spørge om position. Hvis rutinen kaldes med carry sæt,

```
   plot   $fff0    lægges markørens y-værdi i .y og markørens
                   x-værdi i .x, hvor y er søjle, x er linje.
                   Kaldes plot med carry clear, flyttes mar-
                   køren til en plads bestemt ved indholdet i
                   .y og .x, igen med y som søjle, x som linje
            eks.   Flyt markøren til linje 10, plads 5.
                         ldx #10
                         ldy #5
                         clc
                         jsr plot
```

Rens skærmen kan klares ved:

```
   clear  $e544    eller lda #147
                         jsr chrout
```

Brugerporten på C-64.

Hvad du hidtil har programmeret har været smårutiner, der blev vist på skærmen. Der er også en anden måde at sende signaler ud på. Bag på C-64 findes brugerporten. Den rummer bl.a. 8 printbaner, som kan styres med datamaten.
Celle $dd03 rummer dataretningsregister B. Når datamaten tændes, ligger alle bit i denne celle på 0 og stiller dermed alle 8 baner som indgange. Data opsamles i dataregister B, der ligger i celle $dd01. Dataregister B benævnes ofte som port B. Skal port B bruges som udgang, skal dataretningen ændres, d.v.s. $dd03 omstilles til udgang. Det sker ved at lægge $ff i cellen - (alle bit høje). Det er også muligt at stille nogle bit som udgang, andre som indgang. Hvis jeg vil bruge porten til at vise en tæller med ventesløjfe, må det ske således i COMAL:

```
Poke $dd03,$ff // alle bit udgang
for a:=1 to 255 do
  Poke $dd01,a
  For b:=1 to 500 do null // ventesløjfe
endfor a
```

Før du skriver programmet, må du med slukket datamat forbinde stikket med userporten. Når det er sket, kan du tænde og indtaste de få linjer. Start med F-7. Prøv at gemme tælleren med b bag to kommentarstreger og start igen. Kan du se, hvorfor vi må indskyde en forsinkelse ?

Øvelse: Kan du fremstille et program, der får dioderne til at blinke enkeltvis i rækkefølge ?
Se et eksempel her; - findes under "LEDblinker1".

```
POKE $dd03,$ff // alle bit ud
bit:=2
REPEAT
  for a:=0 to 7 do
    poke $dd01,bit↑a
    for c:=1 to 500 do null
  endfor a
UNTIL false
```

Vil vi lave de samme tællinger i maskinkode, må der gøres nogle overvejelser forud:

Gemme tæller i $fb
lave forsinkelse ved at lave to sløjfer,
der gennemløber både .x og .y
- skal stoppe ved et tastetryk.

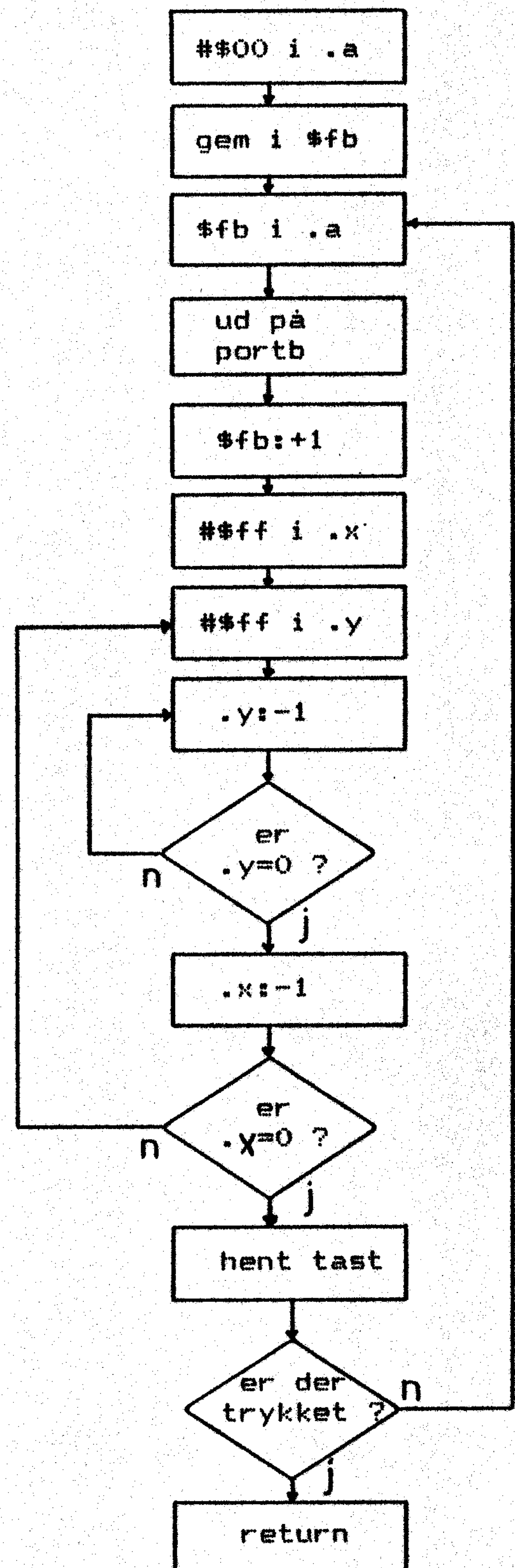
Assemblerkoden findes på næste side sammen med et rutediagram. Forsøg at følge med på diagrammet, når du gennemser koden.

1) Assembler i hånden.
2) Indskriv koden som datalinjer.
3) Få programmet til at køre.

\- o - O - o -

Forstod du opbygningen af forsinkelsessløjfen ? Datamaten står og tæller til 256*256 før den går videre ved at lægge næste tal i $fb.

```
// save "@0:ma.portreg1"
// delete "ma.portreg1"
// * viser tæller ført ud på
// * userport - med forsinkelse
//
adr:=$c000
loop
  read byte
  poke adr,byte
  adr:+1
  exit when eod
endloop
page
//
sys($c000)
//
print at 10,2: "Færdig !"
//
data $a9,$ff //          lda #$ff
data $8d,$03,$dd //      sta $dd03
data $a9,$00 //          lda #$00
data $85,$fb //          sta $fb
data $a5,$fb //          lda $fb
data $8d,$01,$dd //      sta $dd01
data $e6,$fb //          inc $fb
data $a2,$ff //          ldx #$ff
data #a0,$ff //          ldy #$ff
data $88 //              dey
data $d0,$fd //          bne $fd
data $ca //              dex
data $d0,$f8 //          bne $f8
data $20,$e4,$ff //      jsr $ffe4
data $f0,$ea //          beq $ea
data $60 //              rts
```



Denne sløjfe vil vi brodere på lidt senere, idet den skal bruges til vort trafiklys, men først må vi arbejde med nogle nødvendige rutiner.

Trafiklys.
Om opbygningen af printpladerne til trafiklyset kan du læse på side $05 og videre. Der findes også en monteringsplan.

```
// save "@0:ma.portreg1"
// delete "ma.portreg1"
// * viser tællerført ud på
// * userport - med forsinkelsessløjfe
//
adr:=$c000
LOOP
  READ byte
  POKE adr,byte
  adr:+1
  EXIT WHEN EOD
ENDLOOP
PAGE
//
SYS($c000)
//
PRINT AT 10,2: "Færdig !"
//
DATA $a9,$ff //      lda #$ff
DATA $8d,$03,$dd //  sta $dd03
DATA $a9,            lda #$00
                     sta $fb
               hop3  lda $fb
                     sta $dd01
                     inc $fb
                     ldx #$ff
               hop2  ldy #$ff
               hop1  dey
                     bne hop1
                     dex
                     bne hop2
                     jsr $ffe4
                     beq hop3
                     rts
```

Assembler i hånden og indskriv koden i programmet som antydet. - Se diagrammet her ved siden af.

Du husker nok, at de 3 hop skal beregnes som tokomplement.

Næste afsnit, kan godt forbigås foreløbig. Når du senere får brug for bitoperationer, må du tilbage og sætte dig ind i stoffet. Opgaverne fortsætter med trafiklys.

Om bit-operationer.

Begrebet "bit" er nævnt tidligere. Husk at 8 bit danner en byte, der lige netop er den information, der kan lægges i en lagercelle.

Når vi har skrevet

```
lda #$8f
```

er det værdien, der interesserer os, ikke den nøjagtige bitkombination. Er det forresten korrekt, at værdien ovenfor er

%10001111 ?

I statusregisteret har hver bit sin specielle betydning, men der findes også andre tilfælde, hvor de enkelte bit i en "ganske almindelig" byte har særlige, indbyrdes uafhængige betydninger. For eks. kan de 8 bit styre 8 forskellige elektriske kontakter, hvis der sendes et signal ud på brugerporten. Det er denne opgave, der antydes, når vi sætter lysdioder på brugerporten.

Hvis datamaten skal kunne styre kredsløb uafhængigt, må vi finde en måde at sætte og slette de enkelte bit på. Hertil bruges de "Boolske operationer" (efter matematikeren Georges Boole).

Boolske operationer er forbindelser mellem binære tal, og her skal vises AND, OR, EOR og NOT. - Reglerne for disse forbindelser er lettest at forstå, når sandhedstabellerne benyttes bit for bit, når de to binære tal skal sammenlignes.

AND

0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1

OR

0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1

Tabellerne kan læses på flere måder:
Når et 1 og et 0 er forbundet med AND, er resultatet 0. Forbindes de med OR, er resultatet 1. Omvendt kan siges: To bit forbundet med AND kan kun give 1, når den første bit OG den anden bit er 1, ellers bliver resultatet 0. - To bit forbundet med OR, giver 1 når den første ELLER den anden bit (eller begge) er 1, ellers bliver resultatet 0.
Mens man ved AND og OR sætter 2 bit i forbindelse med hinanden, gælder NOT kun for en bit.

NOT 1 = 0
NOT 0 = 1

Foruden skal også nævnes EOR(exclusiv eller), der har denne sandhedstabel:

EOR

0 eor 0 = 0
0 eor 1 = 1
1 eor 0 = 1
1 eor 1 = 0

Resultatet af en forbindelse mellem to byte med EOR, bliver kun 1 hvis den første bit alene (exclusivt) eller den anden bit alene (exclusivt) er 1. Det kan også siges således, at resultatet af EOR (exclusiv OR) er 1, hvis de to bit er forskellige.

Et par eksempler vil støtte forståelsen:

Hvis 2 byte forbindes med AND, må de enkelte bit bearbejdes parvis således:

```
                 1. byte  10110101
        AND      2.   -   11010110
        resultat          10010100
```

resultatbit sættes kun til 1 hvis de tilsvarende bit i 1.byte og 2.byte er 1.

Den kombinationen med OR, kan gennemføres på lignende måde:

```
                 1. byte  10110101
        OR       2. byte  11010110
        resultatet        11110111
```

resultatbit sættes til 1, hvis de tilsvarende bit i 1.byte eller 2.byte er 1. Begge bit kan også være 1.

En EOR-forbindelse mellem de to byte giver:

```
                 1. byte  10110101
        EOR      2. byte  11010110
        resultat          01100011
```

I resultatbyten er bittene kun 1, hvis kun den ene eller kun den anden byte på samme plads er 1.

Der findes assemblerkoder for AND, OR og EOR med følgende mnemonics:

```
        AND
        ORA  (idet alle koder skal bestå af 3 tegn)
        EOR
```

Nu vil vi forsøge at fæstne de lærte med et par eksempler i maskinkode.

HUSK. Ved AND-forbindelser: Resultatet bliver "1", hvis begge celler rummer 1. Hvis en af cellerne er 0, bliver resultatet 0.

En vigtig anvendelse af AND-kommandoen er en metode til at slukke en enkelt bit i en byte.
Hvis f.eks. de fire laveste bit skal nulstilles og de øvrige stå urørt, kan det gennemføres med operationen:

```
1.      lda #$ff
        sta $dd03
        lda #%10101101 ; den byte, der skal kontrolleres
        and #%11110000 ; hermed afmaskes de 4 laveste bit
        sta $dd01
```

Assembler i hånden; indfør maskinkoden i ladeprogrammet og kontroller om resultatet på porten er:

```
        10100000
```

Fremstil et program, der nulstiller 2. og 6. bit, vælg selv den byte, der skal bearbejdes. Herunder er et eksempel:

```
2.      lda #$ff
        sta $dd03
        lda #%11101101 ; den byte du vælger frit
        and #%11011101 ; masken, der skal benyttes
        sta $dd01
```

Er det rigtigt, at dioderne på porten viser:

11001101 ?

Hvad sker der med en undersøgt byte, der afmaskes med %11111111 ?
Lav eksempel 2 om, så du kan afprøve opgaven.

OBS. Hvis den undersøgte byte er

```
         %10101010
   AND   %10111101
   står  %10101000 i akkumulatoren.
```

Denne værdi kan gemmes på en adresse med den sidste kommando, der i eksemplerne er port B.

HUSK ved OR-forbindelser: Resultatet bliver "1", hvis bare en af cellerne rummer 1. Hvis resultatet skal være 0, må begge celler rumme 0.

Anvendes hvis en enkelt bit skal sættes, uden at resten af positionerne påvirkes.
Hvis de 3 højeste bit skal sættes, de øvrige lades urørte, må det ske på følgende måde:

```
3.      lda #$ff
        sta $dd03
        lda #%00101101 ; den byte, der skal behandles
        ora #%11100000 ; de 3 største bit skal sættes høj
        sta $dd01
```

Håndassembler; indskriv i ladeprogrammet, start og kontroller, om resultatet på porten er:

%11101101

Hvad sker der med dette resultat med ora #$00010000 ?
Ret i program 3 og kontroller, om akkumulatoren rummer %11111101

Hvilket resultat giver en OR -forbindelse med $ee ?
Er det korrekt, at bit 2-4 og 6-8 sættes høje ?

Husk: Exclusiv OR eller EOR: Forskellen på EOR og OR er, at kun en af de sammenlignende bit må være 1 for at resultatet af EOR kan være "1"
Konstruktionen bruges under sammenligninger og særlig til fremstilling af det komplementære bitmønster. Dette sker, hvis man EOR en byte med bare et-taller.

```
4.      lda #$ff
        sta $dd03
        lda #%10101010 ; her er den byte, der skal behandles
        EOR #%11111111 ; alle bit er sat høje
        sta $dd01
```

Resultatet på porten er %01010101, idet alle bit i den undersøgte byte er vendt om. Tallet er altså det komplementære udgangstal.

Hvad sker der med et tal ved forbindelsen EOR #$00 ?

Gennemfør en omskrivning af program 4, der viser, om følgende postulat er sandt:

Ved EOR med %00000000, bliver indholdet i akkumulatoren uforandret.

Disse eksempler bruger alle umiddelbar adressering, men alle de nævnte kommandoer tillader derforuden adressering på side 0, x-indekseret side 0, absolut, absolut-x og absolut-y indekseret. I alle tilfælde forbindes akkumulatorindholdet med indholdet af den valgte lagercelle. Resultatet står derefter i akkumulatoren. Forsøg selv at fremstille et par opgaver. Du kan skrive nyt eller ændre i de gennemgåede eksempler, men i alle tilfælde vil det være godt, om du skrev gangen i opgaven i mnemonic-koder.

AND, ORA og EOR er ikke de eneste koder, der kan ændre i bitmønsteret. Der findes også:

ASL - aritmetrisk skift til venstre (forskydning)
LSR - logisk skift til højre (forskydning)
ROR - rotation højreom
ROL - rotation venstreom

I alle tilfælde forskubbes alle bit indenfor lageradressen, men bemærk, at der er 2 forskydninger(shift) og 2 rotationer.
Rotation.

bit nr.	7	6	5	4	3	2	1	0	carry
	1	0	0	1	0	0	1	1	0
ROR	0	1	0	0	1	0	0	1	1

Diagrammet viser en rotation højre om. Læg mærke til, at carry'bit i statusregisteret hele tiden behandles som en slags 9. bit, der flytter med ved operationerne. Ved ROR flytter alle bit 1 plads til højre, carry'bit modtager bit 0 og bit 7 modtager indholdet fra carry'bit.
Forskydning til højre.

bit nr.	7	6	5	4	3	2	1	0	carry
	1	1	0	0	1	0	1	0	1
LSR	0	1	1	0	0	1	0	1	0

Ved forskydning til højre, flyttes alle bit 1 plads til højre, carry'bit arver den udskubbede værdi fra bit 0 og ind fra venstre kommer et 0 til bit nr. 7.
Forskydning til venstre.

	carry	bit nr.	7	6	5	4	3	2	1	0
	0		1	1	0	0	1	1	0	1
ASL	1		1	0	0	1	1	0	1	0

Ved forskydning til venstre, sendes et 0 ind på den tomme plads, mens bit nr. 7 overføres til carry. I eksemplet står $cd i lageret. En forskydning til venstre giver $9a + carry. $cd = 205, - $9a + $100 = 410, altså en fordobling.

Her kommer et lille program, der kan demonstrere forskydning og afmaskning med AND. Fremstil maskinkoden og start programmet. Lav dernæst om i opbygningen, så det kan vise forskydning til venstre og kontroller med afmaskning.

```
5.      lda #$ff
        sta $dd03
        lda #%11110011      ; læg bitmønster ind
        clc                 ; clear mente
        sta $dd01           ; A-reg. ud på porten
        sta $c100           ; gem resultatet
        jsr $ffe4           ; afvent tryk på tast
        beq $fb
        lda $c100           ; hent resultatet
        sta $dd01           ; A-reg. ud på porten
        lsr $dd01           ; forskyd til højre
        jsr $ffe4           ; afvent tryk på tast
        beq $fb
        rol $dd01           ; roter venstre
        and #%10100011      ; afmask med bitmønster
        sta $dd01           ; vis resultat
        rts                 ; til comal
```

Herefter et eksempel, der kan vise rotation til venstre. Inden mnemonic-koden skrives, er det ofte godt for oversigten at lave et diagram over algoritmen.

```
6.      sæt porten som udgang            lda #$ff
                                         sta $dd03
        læg $0a i y-reg.                 ldy #$0a

        læg %00000001 i A-reg.           lda #%00000001

        send A-reg. ud på porten         sta $dd01

        gem A-reg. i $C100               sta $c100

        afvent tryk på en tast           jsr $ffe4

        er der trykket                   beq $fb

        roter venstre $c100              rol $c100

        læg $c100 i A-reg.               lda $c100

        send A-reg. ud på porten         sta $dd01

        y = y-1                          dey

        er y = 0                         bne $ef

        til comal                        rts
```

Det er din opgave at håndassemblere, indskrive koderne i ladeprogrammet og få det til at virke. Lav dernæst om i programmet, så der kan vises rotation til højre.

En vigtig operation, der nu skal omtales er styring af "gemmelager" ved hjælp af stackpointer.
Forestil dig, at du stabler bøger ovenpå hinanden på et bord. Den først lagte bog ligger nederst, den sidst lagte ligger øverst i stakken. Skal en bog tages ud af stakken, må de bøger der ligger ovenover først fjernes - en ad gangen. Dette betyder, at den bog,

der kommer sidst på stakken fjernes først. I princippet fungerer datamatens stak som denne bogstabel. Her drejer det sig bare om lagerceller, der fyldes op med værdier. På vores microprocessor ligger stakområdet fra adresse 256 til 511 - altså hele side 1. Adresse 511 svarer til den underste plads i bogstablen og 256 til den øverste. Der kan altså "stakkes" 128 adresser ad gangen, idet hver adresse bruger 2 byte. I praksis vil der dog ikke kunne være nær så mange.
Ved bogstablen kan du nok se, hvor mange bøger der er, men det er klogt at have en tæller, der forhøjes med 1 hver gang en ny bog lægges på stakken og formindskes med 1 hver gang der fjernes en bog.
I datamaten kan værdierne ikke ses, så her er en tæller nødvendig. Det er denne tæller, der kaldes stackpointeren og den peger på den næste ledige lagercelle. Der findes programstrukturer, hvor en nøjagtig adresse ikke kan angives; - kun data rækkefølgen kendes. Det gælder f.eks. når vi springer til et underprogram.

```
JSR  spring til underprogram (subrutine)
RTS  returnering fra underprogram
```

Ved JSR husker datamaten hvilken adresse, der hoppes fra, så programmet kan fortsætte efter RTS. Hvis der i underprogram 1 kaldes et andet underprogram, lægges den nye adresse på stakken.

Foruden til styring af JSR/RTS findes kommandoer, der direkte kan lægge en værdi på stakken og hente den igen:

```
PHA  skub A-reg. på stak (push accumulator on stack)
PLA  træk A-reg. af stak (pull accumulator from stack)
```

PHA overfører indholdet i akkumulatoren til den første ledige lagercelle i stakken og flytter stackpointeren til næste ledige celle.
PLA flytter stackpointeren en plads tilbage og overfører den senest indskrevne værdi i stakken til akkumulatoren. Samtidig justeres N- og Z-flag.

Skal statusregisteret bevares, sker det på samme måde:

```
PHP  skub statusreg. på stak (push proc.status on stack)
PLP  træk statusreg. af stak (pull proc.status from stack)
```

Efter gennemgang af disse kommandoer, er det muligt at forbedre programeksempel 5. Læg kommandoen PHP ind lige før afvent tryk på taste og PLP ind lige før hent resultatet. Derved kan vi nemlig bevare statusregisteret, selvom der sker ændringer med registrene, når der ventes på et tryk på en taste. Lav denne tilføjelse og få programmet til at virke.

Vender vi et øjeblik tilbage til ASL, LSR, ROL og ROR, kan de adresseres i akkumulator, x-indiceret zeropage, absolut, absolut indiceret. Læg mærke til, at resultatet af operationen står i det register eller i den lagercelle hvor flytningen eller rotationen foregår. Skal man bruge carry'bit til en kontrol, må man lægge statusregisteret på stakken med PHP og senere hente det ind igen med PLP, hvis der skal afvikles andre opgaver inden kontrollen gennemføres. Z-flag og N-flag sættes eller slettes i overensstemmelse med operationen. Carry-flaget bliver som nævnt bruge som en mente, en 9.bit.

Efter at vi nu har været en del kommandoer igennem som gør det muligt at ændre en byte bit for bit, skal vi have fat på en kom-

mando, der tillader bitsammenligninger. Normalt bruges CMP (compare), CPX (compare x) eller CPY (compare y) til sammenligninger, men de egner sig kun til tal, - ikke til sammenligning af enkelte bit. Her skal bruges :

BIT, der udfører et AND mellem en lagercelle og A-reg.

Den laver ikke om på indhold i lagercelle eller akkumumator; resultatet går tabt, men statusregistret opdateres af resultatet. BIT har indflydelse på statusflagene Zero, Negativ og V overflow. De mulige adresseringsmetoder er zero-page og absolut. Efter BIT følger så godt som altid en branch-kommando. Her er et eksempel: Vi tænker os at tastaturet lægger en kode i en lagercelle, vi tager $fe på side 0. Nu skal vi skrive den lille del af programmet, der venter på at en bestemt talværdi opnås. Tælleren lægger indholdet i $fe og vi er færdige, når tælleren er nået til 8, altså når bit nr. 3 sættes. De andre bit er vi ikke interesserede i. Vi venter altså på bitmønsteret %00001000 og må lave en AND-sammenkædning, hvor kun flagene sættes efter resultatet.

```
.
.
lda #%00001000    ; bitmønster lægges i A-reg.
bit $fe           ; der sammenlignes med $fe
beq $fc           ; sålænge Z=0 springes -4 pladser
.
.
```

Sålænge tælleren ikke er nået til 8, er bit nr. 3 i $fe 0, BIT $fe er altså 0 og Z-flaget er sat. - Der skal altså springes 4 pladser tilbage for ny sammenligning. Når tælleren når 8, bliver bit nr. 3 i $fe sat, resultatet 1 vil lægge Z-flaget og programmet fortsætter.

Kommandoen kaldes også for BIT-testeren. Behandling af flagene ses også af følgende eksempel. Vi kommer ind i programmet, hvor der står:

```
.
lda #%01000000    ; bit nr. 6 afmaskes
bit $dd01         ; sammenlign med port b
.
```

Hvad der sker, kan forklares således: Indholdet i celle $dd01 sammenlignes med A-reg. indhold gennem et logisk AND. Resultatet påvirker Z-, N- og V-flag, hvorimod hverken indhold i celle $dd01 eller A-reg. ændres. Bit for bit AND-forbindes A-reg. og $dd01's indhold. Hvis de undersøgte bits i begge komponenter er 1, er resultatet 1

```
A-reg.     % 01000000
$dd01      % 01001010
AND
resultat     01000000
```

I eksemplet har vi med BIT-testen villet kontrollere, om bit nr. 6 i $dd01 er slukket. Derfor lægges en maske i A-reg. Hvis bit 6 er slukket, vil resultatet være 0 og Z-flaget dermed sat. Gennem en maske er det altså muligt at undersøge et enkelt bit (eller flere). - Men der sker mere: Bit nr. 6 og bit nr. 7 i den undersøgte celle genfindes i status-reg.

bit nr. 7 i N-flaget (negativ)
bit nr. 6 i V-flaget (overflow)

Hermed kan man undersøge, om der i cellen ligger et negativt tal (bit 7 sat). Alle 3 flag kan undersøges med en branch-kommando.

CLV betyder CLear oVerflow
NOP betyder No OPeration

Det eneste der her sker er, at programtælleren går 1 skridt frem og denne operation tager 2 taktslag. NOP anvendes sjældent i færdige programmer, men kan benyttes som "pladsholder" under udarbejdelsen af et program.

Hvad sker der, når dette program afvikles ? Du kan håndassemblere det og indføre det i ladeprogrammet og starte det.

```
        lda #%00111111
        sta $dd03
mærke   lda #%00111111
        sta $dd01
        lda #%01000000
        bit $dd01
        bne mærke
        rts
```

Hvad vil der ske, hvis næstsidste linje ændres til beq mærke ? Prøv og se, om din antagelse er rigtig. Du kan bruge dine prøveplader til opgaven.

Herefter går vi ind i opgaverne igen.

- o - O - o -

Her er en repetition i brug af subrutiner. Læs om færdige rutiner, side 34.
Fremstil et maskinkodeprogram, der sletter skærmen, stiller markøren på 10. linje, 5. plads og venter på at du indtaster et bogstav. Når det er sket, skal bogstavet vises på skærmen og dets ASCII-værdi vises på din plade med lysdioder. Bogstavet skal stå på skærmen i den tid det tager for programmet at tælle til 0 fra 256*256; - derefter vendes tilbage for nyt forsøg. Tryk på <F-1> får programmet til at stoppe.
Du kan benytte getin, chrout, plot og clear. En subrutine der venter, må du selv skrive. Måske bør der stå lidt vejledning på skærmen.

Eksempel "src.asciiskærm" på disketten.

Husk ! : Du skal bruge assembler c-64 for at hente kildeteksten ind i datamaten. I eksemplet findes også en rutine med x-indexeret adressering. Denne metode vil blive behandlet mere indgående senere.

Hvordan vil du kontrollere om den oplysning, der står på porten er korrekt ?

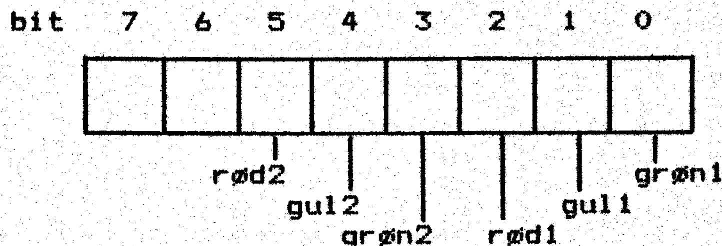
Trafiklys, styret i maskinkode.

Hertil skal du bruge printpladerne, der sammenbygget danner et sæt trafiklys.
Først vil vi lave en subrutine kaldet "vent". Den skal hente en tæller fra celle $fb, trække 1 fra denne tæller og derefter gennemløbe 2 sløjfer og springe retur, når tælleren er 0. Gennemløbet af de to sløjfer varer meget nær 0,1 sek. Det tal, der lægges i $fb kan altså bestemme forsinkelsen.

```
vent  ldx #$9d   ; læg $9d i .x
mk2   ldy #$71   ; læg $71 i .y
mk1   iny        ; .y:+1
      cpy #$00   ; er .y=0
      bne mk1    ; hvis nej - hop til mk1
      inx        ; .x:+1
      cpx #$00   ; er .x=0
      bne mk2    ; hvis nej - hop til mk2
      dec $fb    ; gør indh. i $fb 1 mindre
      lda $fb    ; hent indh. i $fb
      cmp #$00   ; er det =0
      bne vent   ; hvis nej - start forfra
      rts        ; retur til hovedprogram
```

Et enkelt gennemløb fra 255 til 0 sker for hurtigt til vores brug. Fremstilles 2 sløjfer inden i hinanden, så begge løber fra 255 til 0, skal datamaten tælle til 256*256 inden den går et tællertal frem. I vores eksempel startes med $9d i den yderste sløjfe og $71 i den inderste. Det tager meget nært 0,1 sek at komme dette forløb igennem. Tælleren i $fb danner en tredje sløjfe og her tælles nedad til $00, hvorefter der springes tilbage til hovedprogrammet. Hvis vi lægger $0a i $fb, vil det tage 1 sek at udføre rutinen.

Fordelingen af røde, gule og grønne LED:



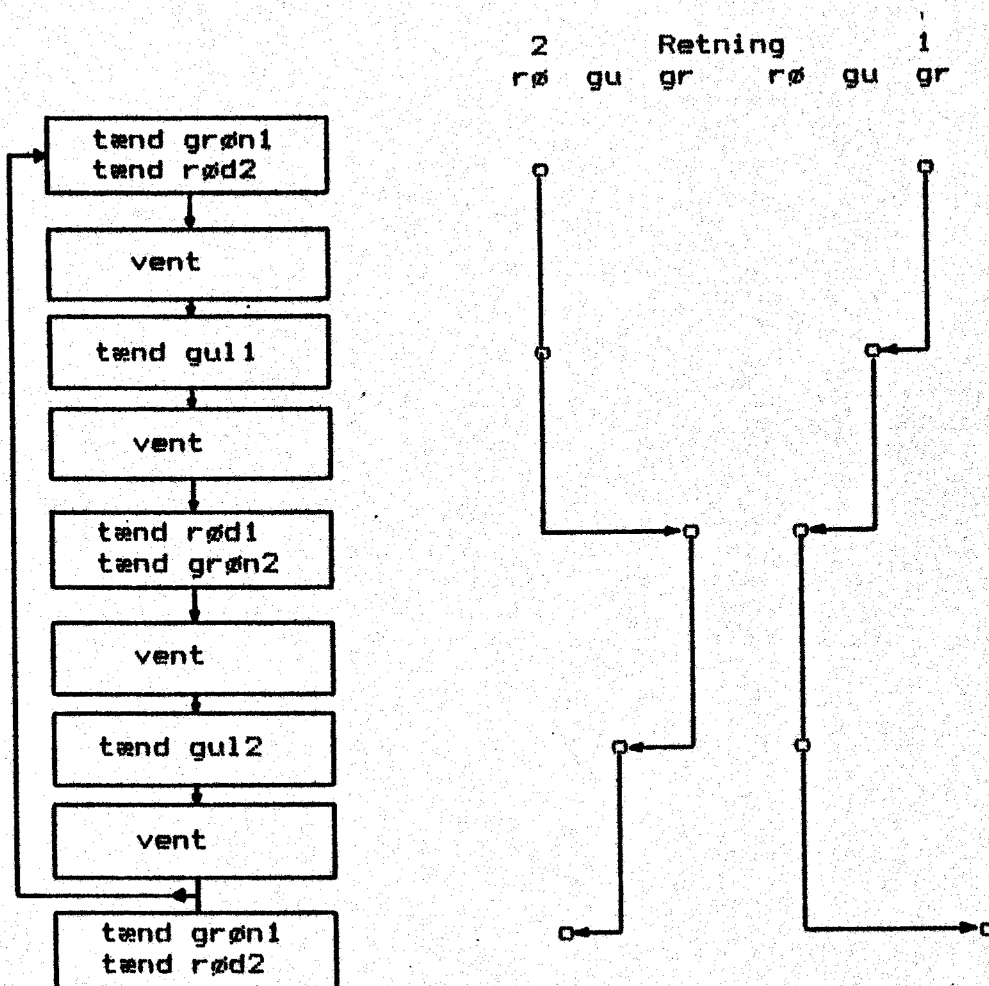
Fordelingen af byte ses på tegningen:

lampe	bitmønster	hex
grøn1	%00000001	$01
gul1	%00000010	$02
rød1	%00000100	$04
grøn2	%00001000	$08
gul2	%00010000	$10
rød2	%00100000	$20

Først vil vi lave et nattelys, der blinker gult i begge retninger, skiftevis. Skemaet for denne opgave ses på næste side:

```
nat   lda #$3f      ; %00111111                tænd gul1
      sta $dd03     ; dataretregB              sluk gul2
nat2  lda #$02      ; tænd
      sta $dd01     ; gul1
      lda #$0a      ; tæller 10
      sta $fb       ; tællercelle                 vent
      jsr vent
      lda #$10      ; tænd
      sta $dd01     ; gul2                     tænd gul2
      lda #$0a      ; tæller 10                sluk gul1
      sta $fb       ; tællercelle
      jsr vent
      jsr $ffe4     ; hent tegn                   vent
      bne ud1       ; tast: stop
      jmp nat2      ; forfra
ud1   rts
```

Kan du forklare, hvordan du får lagt tælleren ind i celle $fb ? – Vi skal også kunne bruge lyskurven om dagen. Derfor går vi videre med dagrutinen.



Det er et superhurtigt kryds, hvor vi vil se på det enkelte forløb. Retning2 er rød mens retning1 gennemløber grøn og gul. Vi ønsker 7 sek. kørsel, 3 sek. gul, 7 sek. kørsel i modsat retning, 3 sek. gul. Denne plan skal være vort udgangspunkt.

```
dag   lda #$3f   ; %00111111
      sta $dd03  ; dataretningB
dag1  lda #$21
      sta $dd01  ; grøn1+rød2
      lda #$46   ; decimal 70
      sta $fb    ; tællercelle $fb
      jsr vent
      lda #$22
      sta $dd01  ; gul1+rød2
      lda #$1d   ; decimal 30
      sta $fb    ; tællercelle $fb
      jsr vent
      lda #$0c
      sta $dd01  ; rød1+grøn2
      lda #$46   ; decimal 70
      sta $fb    ; tællercelle $fb
      jsr vent
      lda #$14
      sta $dd01  ; rød1+gul2
      lda #$1d   ; decimal 30
      sta $fb
      jsr vent
      jsr $ffd2
      bne ud2
      jmp dag1   ; forfra
 ud2  rts
```

Hele styreprogrammer må være nat+dag+vent. Du skal ikke skrive dem ind endnu; - men det kommer ! - Gennemgå de 3 dele med din makker og forklar ham gangen i programmet. Det er en god ide at støtte sig til blokdiagrammerne.

Vi er nu nået op på en programstørrelse, hvor det vil være klogt at finde en anden mulighed end at assemblere i hånden. Heldigvis er der mange programmer, der kan klare opgaven at oversætte fra mnemonic-kode til hexkode. En sådan oversætter kaldes også en assembler. Det er altså et oversætterprogram, vi har brug for. På disketten findes en god assembler med navnet "ASSEMBLER C-64". Her er der naturligvis en del kommandoer og et regelsæt, som skal overholdes, hvis programmet skal fungere korrekt. Derfor følger en kort indføring i hvordan et assemblerprogram bør opstilles:

Om assemblerprogrammet:

I eksemplerne forud har vi allerede brugt op.koder, der hører til 65xx assemblersproget. Koderne har vi via kodetabellen oversat til hexadecimale tal, som mikroprocessoren kan forstå. Ved lidt større opgaver er dette arbejde langsommeligt og svært at overskue. Derfor er der udviklet et program, der kan klare opgaven for os. - Dette program kaldes også en assembler.

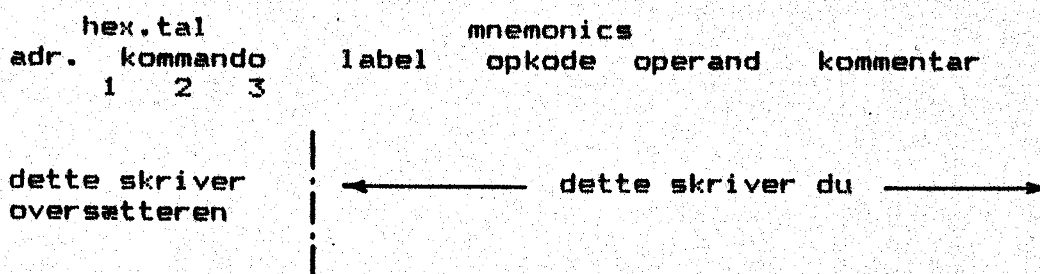
Altså: Sproget med mnemonickoder kaldes assembler. Et program skrevet i mnemonickode kaldes en kildetekst, en sourcekode. Filnavne herfra bør starte med SRC.*******.

Programmet der oversætter kildeteksten til maskinkode kaldes også en assembler. Dette program kan hente en kildetekst fra diskette, - oversætte den til maskinkode og gemme denne kode på diskette som en objectfil. Filnavne bør starte med OBJ.*******.
Samme program anvendes også når kildeteksten skrives, idet det indeholder en editor, der muliggør rettelse af fejl under arbejdet.

Alt dette sker i BASIC. Den assembler jeg vil omtale i dette afsnit er nem at arbejde med, idet kommandoerne ligner COMALS. Den har dog det minus, at den ikke direkte kan benytte COMALS symboltabel. Den fil, du ved - på demonstrationsdisketten "c64symb". De anvendte symboler må findes og skrives i en symbolliste i begyndelsen af kildeteksten.

Her kommer introduktionen:

En kildetekst er opbygget i flere søjler, som det er vist her på skemaet.



Hex.koden fremstilles af programmet. Vi må skrive mnemonic.koden og det hører til god skik, at der skrives fyldige kommentarer til koderne. 2 måneder efter at koden er skrevet, kan du ikke huske de geniale indfald du benyttede.

label: Her kan stå et mærke, der bruges som mål for spring
opkode: Er 3-bogstavskoden, der efterfølges af en
operand: Kan opgives decimalt, hexadec.($) eller binært(%)

kommentar: Sørg for fyldige bemærkninger; de kommer ikke med i objektkoden.

En stor lettelse er brug af symboler. I begyndelsen af kildeteksten kan du lave en liste over symboler og de tilhørende adresser (symboler max. 7 tegn). Du kan ikke benytte store Æ, Ø, Å eller lille æ, ø, å i navne på symboler eller i operander; - i kommentarfeltet er det derimod tilladt.

Hvis du f.eks. skriver:

```
skaerm=$0400        ; skærmstart
dataret=$dd03       ; dataretningsreg
portb=$dd01         ; dataportB
tast=$ffd2          ; hent tast
```

kender assembleren disse adresser efter 1. gennemløb. Når du skal bruge dem i programmet, skrives symbolet som operand, og programmet udregner selv de korrekte adresser.

En label har ingen talværdi tildelt. Under assembleringen får den tilknyttet den adresse den følgende opkode har.
Konstanter indledes med #
Psudokoder er anvisninger som assembleren selv skal bruge for at kunne behandle kildeteksten efter ønske.
Når assembleren behandler kildeteksten, sker det ved to gennemløb. Efter 1. gennemløb kender programmet startadresse, symboladresser, labels m.v. og der er udført kontrol for evt. syntaxfejl. Under 2. gennemløb regner programmet selv frem til adresser, der skal benyttes af programtælleren senere, udformer hex.koderne, udregner spring til og fra underrutiner m.v. - Endelig fremstilles tabel over variabler og symboler.
Er der fejl i kildetekstens syntax, stoppes og du får angivet fejlens placering. Efter rettelse, kan du assemblere påny uden tab af koden. Hvis afviklingen er kommet så lang før en fejlmelding indtræffer, at objektkoden er begyndt at komme på disketten, må du enten slette den gamle fil eller give objektkoden et nyt navn før genstart.

- o - O - o -

Assembler C-64. En vejledning, der også kan bruges privat.

Start. Assembleren hentes ind i datamaten og startes med:

```
Basic <RETURN>
LOAD "ASSEMBLER C-64" ,8,1  <RETURN>
SYS 64738 <RETURN>
```

Herefter melder assembleren sig på skærmen med angivelse af, at der er 30 Kbyte til rådighed. De ligger i arbejdslageret fra $0801 til $7fff. Der er oprettet et specielt lager for labels og andre hjælpetekster og området $c000 til $cfff er holdt frit, så en monitor, f.eks. SMON kan være i datamaten samtidig.

Kildeteksten (src=sourcecode) indtastes som et basicprogram, d.v.s. at der kan rettes, slettes og tilføjes. Hver linje har et nummer og hver linje har følgende indhold:

linjenr. evt.label   mnemonic-kode   lagercelle m.v.; kommentar

Der gælder følgende regler for indtastningen:

1. hver linje må have et entydigt linjenummer; AUTO- forefindes.
2. uden mellemrum følger herefter en evt. label (ingen tom plads).
3. assemblerbefaling adskilles fra label med et mellemrum.
4. hvis en linje ingen label rummer, begyndes med en tom plads.
5. en kommentar starter med et semikolon.
6. en ren kommentarlinje starter som 2.
7. hvis en linje kun rummer en label, må der ingen kommentar stå.

De normalt kendte kommandoer OPEN, CLOSE, CMD, PRINT#, PEEK, POKE og SYS kan anvendes, men der må ikke tildeles værdier gennem FOR, NEXT, DIM.
Derforuden findes en serie editorkommandoer, der gennemgås herunder.

Editorkommandoer.

A (auto)   A linjenr.,skridt   eks.: A 100,10
Giver automatisk linjenummerering, der kan afbrydes med <SHIFT><RETURN>. Nummereringen kan genoptages med A <RETURN>. Kendes fra COMAL.

B (lager til rådighed) <F-2>
Viser den ledige lagerplads i Kbyte.

C (koldstart)
Starter assembleringen pány. Det er ingen reset. Med O <RETURN> hentes kildeteksten tilbage.

D (delete)   D linjenr.1 - linjenr.2 eks.: D 500-560
Undlades linjenummer, slettes hele teksten, men der stilles et sikkerhedsspørgsmål først.
D -100 sletter til og med linje 100
D 250 sletter kun linje 250

E (list) <F-1> E linjenr.1 - linjenr.2 eks.: E 710-820
Viser kildeteksten. Listen kan standses med mellemrum og genoptages på samme måde. Funktionen afbrydes med RUN/STOP.
Udlistning på papir sker med OPEN 4,4,7:cmd4 <RETURN> E <RETURN>

I (indhold) <F-7>
Viser indholdsfortegnelsen for den diskette, der ligger i drev 8. Kan stoppes med tryk på <mellemrum>

L (load) <F-5>   L "programnavn",8
Henter en kildetekst fra diskette og lægger den i arbejdslageret fra $0801. Vælges LOAD "programnavn",8,1 lades programmet absolut.

M (merge)   M "programnavn"
Henter en kildetekst og lægger den i forlængelse af en anden tekst. Her er altså mulighed for at hægte tekster sammen og assemblere dem under et. Det må være en selvfølge, at variabelnavne har samme definition.

N (nummer)   NO,startnummer,skridt   eks.: NO,100,5
Renummererer teksten som det kendes fra COMAL.

O (old) <F-6>
Henter en gammel kildetekst tilbage efter reset. Se også C.

R (replace)   RO,"nytord","gammeltord"
Kommandoen søger efter gammeltord og ombytter det med nytord.

S (save)   S "programnavn"
Kildeteksten saves som programfil på diskette. Hvis man i forvejen har brugt LOAD for at hente en monitor, kan programpointeren stå forkert. Her anbefales at taste O <RETURN> og derefter S <RETURN>.

T (tabulator)   eks. T0,10 T1,24
  Indstiller tabulator 0 og tabulator 1, der har betydning for udlistningen. T0 bestemmer hvor mange pladser en label må fylde. T1 angiver, hvor kommentaren kan starte. Normalindstillingen ses ovenfor i eksemplet.

V (verificer) V "programnavn"
  Svarer til basickommandoen

X (start assemblering) <F-3>
  Her kan også bruges RUN <RETURN>. Kildeteksten gennemløbes 3 gange, hvor labels og lignende bemærkes. Selve udskrivningen og fremstilling af objektkode sker under 3 gennemløb.

Y (udskriv symboltabellen)
  Efter assemblering vises symboltabellen på skærmen. Skal den listes ud på printer, er kommandoen:

```
OPEN 4,4,7:CMD 4:Y <RETURN>
OPEN 4,4,7:CMD 4:LIST <RETURN>  bruges også. Buffer tømmes med
PRINT#4:CLOSE 4 <RETURN>
```

NB. Skal en linje rettes, bør hele linjen skrives om.

Pseudokoder.

Disse koder er anvisninger til assembleren om at foretage sig bestemte operationer under løsning af opgaven. De indledes alle med punktum.

.GLOBAL definerer en label, så den er kendt i hele programmet
```
      .global cwrt=$ca06
      .global chrout=$ffd2
```
herefter kendes rutinen, der starter i $ca06 som cwrt og rutinen fra $ffd2 som chrout.

.EQATE definerer en label lokalt. Den kan ikke bruges i macros.
```
      .equate getlin=$ca22
```
lokale labels kan bruges til sløjfer og til springkommandoer.

.BYTE indpasser en enkelt byte i kildeteksten. Værdien ligger mellem $00 og $ff. Flere byte kan stå på samme linje, hvis de er adskilt med komma.
```
      .byte 32, $20, %10000000, " " angiver 4 mellemrum.
```
Der kan bruges decimale-, hexadecimale-, binære tal og strenge med 1 plads.

.WORD indpasser en adresse, altså en 16-bit værdi i kildeteksten.
```
      .word hilsen = lille byte hilsen, store byte hilsen
```
Der benyttes normal skrivemåde for lageradresser og flere word kan stå på samme linje, hvis de adskilles med komma.
Lille byte/store byte af en adresse kan også angives således:
```
      <($0400), >($0400)          lille byte/store byte

      eksempel:  lager=$c108
      ldx #<(lager) ; læg lille byte lager i .x   ($08)
      lda #>(lager) ; læg store byte lager i .a   ($c1)
```

.DS reserverer pladser i programmet til senere deltekster.
```
      .ds 38 reserverer 38 byte lagerceller til senere brug
(max.255)
```

.TEXT tillader indføjelse af tekst i kildeteksten. De enkelte tegn aflægges som ASCII-koder.

.text "bogstaver" med afsluttende anførselstegn indføjer et 0

Derved kan teksten udskrives med rutinen $able (i basic)

.text "bogstaver lægges uden 0-byte

Er det nødvendigt med slutmærket, kan man indføje .byte $00

Tasten <pil venstre> giver en vognretur i teksten. Samme funktion opnås ved at benytte .byte 13

.OBJECT leder objektkoden til diskettestationen

.object "programnavn,p,w"

sender assemblerkoden direkte til diskettestationen. Her skal kildeteksten sluttes med .END for at få filen lukket korrekt. Hvis dette glemmes, kan man selv indføje med direkte kommando:

close 14 <RETURN>

.BASE fastlægger startadressen på objektkoden.

.base $c009

.SYMBOLS bevirker at der efter assembleringen udgives en symboltabel.

.symbols 4,4,7 udskriver på printer, hvis datakanal 4 er åbnet

.symbols 1,3,0 viser på skærm (kan holdes med <mellemrum>)

.symbols 2,8,2,"comalsymb.ass64,s,w" skriver til disketten

.LISTING fremstiller en udprintning af det assemblerede program under opgavens løsning.

.listing 4,4,7 "programnavn"

OBS. Ønskes både en listning og en symboltabel, må de udskrives ad hver sin datakanal.

.END afslutter og lukker objektfilen

.STOP forårsager afbrydelse i assembleringen, dog først i 3 gennemløb. Dermed sikres, at alle labels står til rådighed. Objektkoden fremstilles kun til stopmærket.

.PAGE sender sideskift til printeren således:

.page 60 bevirker sideskift efter hver 60. linje.

.NOCODE forhindrer at objektkoden lægges i lageret, hvilket er en fordel ved skrivning til disketten.

.START Hvis denne kommando findes, springes efter assembleringen til denne adresse, skærmen renses med <mellemrum> og det fremstillede program prøvekøres.

OBS. programmet må slutte med RTS, så der springes tilbage til assembleren.

Her er et eksempel på brug af ASSEMBLER C-64.

Når programmet er hentet ind i datamaten og startet, skal vi til at skrive de enkelte linjer i assembler. Se nøje på reglerne nederst side 50. Som eksempel vil vi bruge programmet side 36. Her tæller datamaten på userporten med en forsinkelsessløjfe.
Tast A 100,10 <R> og du er klar til at begynde. Udlistningen herunder har fyldige kommentarer. Når du har skrevet kildeteksten, bør den saves med navnet "src.programnavn",8 <R>.
Er alt klart til assembleringen: Diskette i diskettestationen og printeren tændt, tastes X <R> og opgaven bearbejdes. Resultatet kan ses på næste side.

```
100; eksempel på kildetekst
110; portreg1, maskinkodedelen, den
120; skal hægtes på ladeprogrammet
130; i comal. version okt.88
140;
150           .object "obj.prøveport1,p,w"
160           .symbols 4,4,7
170           .listing 5,4,7
180;
190; her angives programstart
200;
210           .base $c000
220;
230; variabler defineres
240;
250           .global retreg=$dd03
260           .global portb=$dd01
270           .global get=$ffe4
280;
290; her begynder assemblerprogrammet
300;
310           lda #$ff      ; læg 255 i .a
320           sta retreg    ; porte som udgang
330           lda #$00      ; læg 0 i .a
340           sta $fb       ; gem indhold i celle $fb
350mk1        lda $fb       ; hent indhold fra $fb
360           sta portb     ; læg ud på porten
370           inc $fb       ; opskriv indhold i $fb med 1
380           ldx #$ff      ; læg umiddelbart 255 i .x
390mk2        ldy #$ff      ; læg umiddelbart 255 i .y
400mk3        dey           ; nedskriv .y med 1
410           bne mk3       ; er .y ej 0 så hop
420           dex           ; nedskriv .x med 1
430           bne mk2       ; er .x ej 0 så hop
440           jsr get       ; er en tast trykket ?
450           beq mk1       ; hvis nej, så hop
460           rts           ; retur til comal
470           .end

assembler klar
```

Her ses resultatet af assemblerens arbejde. Du kan kende kildeteksten fra din egen indtastning, men derforuden er der også skrevet de tilhørende byte som danner maskinkoden. De ligger også i en programfil på disketten med navn "obj.prøveporti". - Om den videre behandling kan læses på næste side.

```
line #  loc    code      line
  180                    ;
  190                    ; her angives programstart
  200                    ;
  210                    .         .base $c000
  220                    ;
  230                    ; variabler defineres
  240                    ;
  250          #= 03 dd            .global retreg=$dd03
  260          #= 01 dd            .global portb=$dd01
  270          #= e4 ff            .global get=$ffe4
  280                    ;
  290                    ; her begynder assemblerprogrammet
  300                    ;
  310   c000   a9 ff               lda #$ff       ; læg 255 i .a
  320   c002   8d 03 dd            sta retreg     ; porte som udg
  330   c005   a9 00               lda #$00       ; læg 0 i .a
  340   c007   85 fb               sta $fb        ; gem indhold i
  350   c009   a5 fb     mk1       lda $fb        ; hent indhold
  360   c00b   8d 01 dd            sta portb      ; læg ud på por
  370   c00e   e6 fb               inc $fb        ; opskriv indho
  380   c010   a2 ff               ldx #$ff       ; læg umiddelba
  390   c012   a0 ff     mk2       ldy #$ff       ; læg umiddelba
  400   c014   88        mk3       dey            ; nedskriv .y m
  410   c015   d0 fd               bne mk3        ; er .y ej 0 så
  420   c017   ca                  dex            ; nedskriv .x m
  430   c018   d0 f8               bne mk2        ; er .x ej 0 så
  440   c01a   20 e4 ff            jsr get        ; er en tast tr
  450   c01d   f0 ea               beq mk1        ; hvis nej, så
  460   c01f   60                  rts            ; retur til com
  470


                                   .end

symboltabel :

line   250: retreg.........$dd03 global
line   260: portb..........$dd01 global
line   270: get............$ffe4 global
line   350: mk1............$c009      0
line   390: mk2............$c012      0
line   400: mk3............$c014      0

assembleringstid   00:33:3
object range $c000 - $c020

src.tekst fylder    887 bytes (0 k)
obj.kode fylder      32 bytes (0 k)
symb.tabel fylder    42 bytes (0 k)
```

Hvis du vil fortsætte i BASIC, kan programmet hentes ind med kommandoen LOAD"OBJ.PRØVEPORT1",8,1 <R>, hvor "1" bevirker, at programmet lægges på den rigtige plads. Herefter startes med SYS 49152 <R>.

- o - O - o -

Da vi hellere vil benytte COMALS gode egenskaber, må vi omdanne objectkoden til en seq-fil, der kan hægtes på et ladeprogram. Et sådant findes i flere udgaver på disketten.
Slå om til COMAL ved at resette datamaten og hent så programmet "lav'data'linjer" ind. Start det og indtast navnet på den prg-fil, der skal omdannes. Programmet skal også bruge et navn til den nye seq-fil med data, men det giver selv filen fornavnet "dat."
Efter afviklingen ligger der en datafil, der kan merges til et ladeprogram. Når dette er sket, mangler vi bare at gemme datalinjerne startadresse og kontrolsum bag et par kommentarstreger. - Se eksemplet herunder.
Alle 4 programmer findes på disketten, men prøv selv at gennemføre opgaven på din egen diskette.

```
// save "@0:prøveport1"
// delete "prøveport1"
// * viser tæller ført ud på
// * userport - med forsinkelse
// * til ladeprogrammet er knyttet
// * datafil med kommando "merge"
//
adr:=$c000
LOOP
  READ byte
  POKE adr,byte
  adr:+1
  EXIT WHEN EOD
ENDLOOP
PAGE
//
SYS ($c000)
//
PRINT AT 10,2: "Færdig !"
//
//DATA $c000 // start adresse
DATA $a9,$ff,$8d,$03,$dd,$a9,$00,$85
DATA $fb,$a5,$fb,$8d,$01,$dd,$e6,$fb
DATA $a2,$ff,$a0,$ff,$88,$d0,$fd,$ca
DATA $d0,$f8,$20,$e4,$ff,$f0,$ea,$60
//DATA $168e // kontrolsum
END
```

Nu har du prøvet at skrive en kildetekst, der er endt som maskinkode i lageret. Det kunne nok være rart at råde over et program, hvormed vi kunne kigge i lageret, indskyde tilføjelser, flytte om på placering, disassemblere og udføre andre opgaver - eller bare undersøge maskinkoden.
Hertil må vi benytte en monitor. Dem findes der mange af, - og her vises en vejledning, der knytter sig til SMON. Dette program kan klare alle vore opgaver uden at det bliver for indviklet. Selve vejledningen findes på næste side; - men programmet må anskaffes via Markt & Technik, Hans Pinsel Strasse 2, D-8013 Haar, BRD. Dette firma udgiver bladet "64'er", der også er kendt her i Danmark.

Hvis tæller A og tæller B sammenkobles, B tæller underflow fra A.
Reg.B %01010001 = $51 // underflow, reset, start
Reg.A %00010001 = $11 // reset og start
Stop (A og B) %00000000 = $00 // til begge registre

Hvis du henter det tidligere program med tælling, kan det ombygges til at vise indholdet i de 4 registre. Det mindste register og antagelig det næstmindste register må behandles med omtanke. På trods af COMALS gode egenskaber, er programafviklingen så "langsom", at disse registre ikke altid vises korrekt.

```
USE system
page
print at 5,5: "Tast.........."
while key$=chr$(0) do null
time 0
poke $dd0f,%01010001 // tæl underflow, reset B, start
poke $dd0e,%00010001 // reset A, start
while time<3600 do null
poke $dd0e,%00000000 // stop A
poke $dd0f,%00000000 // stop B
print at 7,5: time
print at 9,5: "Færdig !"
print "$dd07 B store byte :";peek($dd07)
print "$dd06 B lille byte :";peek($dd06)
print "$dd05 A store byte :";peek($dd05)
print "$dd04 A lille byte :";peek($dd04)
```

Undersøg på egen hånd programmet med fi-2 clock. "src.jiffies" og "jiffies"
Hvad kan det vise ?

De gennemgåede eksempler findes på disketten som "tællerforsøg" 1 - 5.

- o - O - o -

Ekstraopgaver:

a) Ombyg "ma.adition2" så prøvepladen med lysdioderne kan vise resultatet. Det kan ordnes således, at den viser lille byte, venter på tast, viser store byte, venter på tast og vender tilbage til COMAL og viser resultatet på skærmen.

b) Omskriv forsinkelsessløjfen i "LEDtæller1" så der decrementeres i yderste sløjfe og blinkes dobbelt så hurtigt.

c) Omskriv forsinkelsessløjfen i lyskryds så der incrementeres. Beregn startværdierne, der skal lægges i .x og .y (husk at rette både i sløjfe og i hovedprogram).
Hvilke fordele har denne løsning ?

d) Omskriv hovedprogrammet i lyskryds, så lamperne tændes v.hj.a. en EOR-kommando. Se evt. først under bitoperationer.
Hvilke fordele har denne løsning ?

Mere om tællere og phi-2 klokken.

Phi-2 klokken styrer også 3 registre på side 0. Herfra hentes oplysningerne, hvis du skriver PRINT TIME <R>. Tallet der fremkommer på skærmen er angivet i jiffies. En mere komfortabel metode er fra systempakken at kalde funktionen gettime$. Her vises tiden fra nulstilling i format hh:mm:ss.j. Oplysningerne ligger i adresserne $a0, $a1, $a2. Disse adresser er koblet sammen som omtalt under tællere.

```
USE system
settime("0") nulstiller uret
print gettime$ giver tiden fra nulstilling
```

Du kan også bestemme udskriften

```
settime("hh:mm") indstiller til valgt time og minut
```

I formatet hh:mm:ss.j er j tiendedele sekunder.

Som vi senere skal se, kan et maskinkodeprogram, formet som en pakke konstant aflæse tælleren og vise resultatet på en valgt plads på skærmen som hh:mm:ss. Se "uret.skærmclock" - den kan du også bruge i egne programmer. Programmet "jiffies" skal have et par ord med på vejen. Formålet med dette program er at finde det antal skridt phi-2 klokken går frem på 1 sek. Vi skal altså nøje afmåle 1 sek og derefter i comal aflæse tælleren. Det sker således:

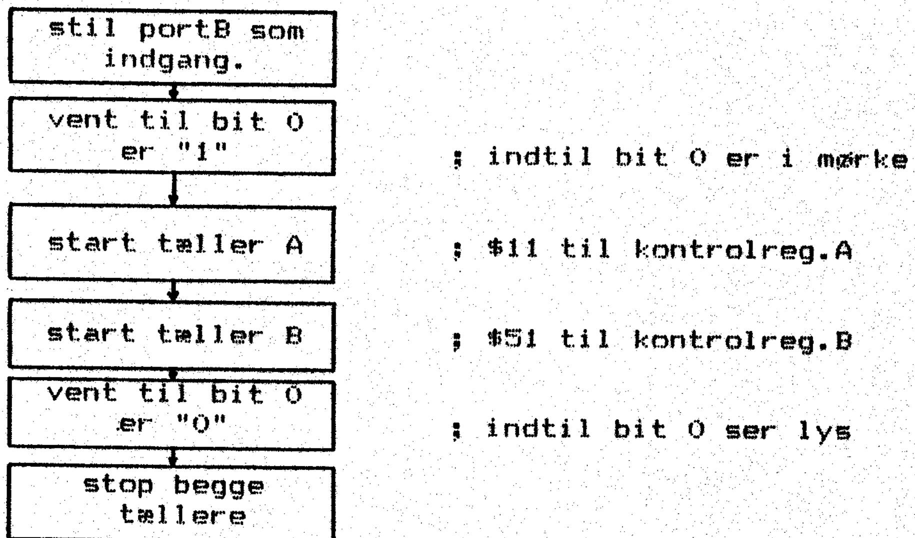
```
     lda #$51       ; startværdi for tællerB
     sta $55        ; gemmes på side 0
     ldx #$00       ; stopværdi gemmes i .x
     lda #$11       ; startværdi for tællerA
mk1  ldy $a2        ; hent lille byte jiffies
     bne mk1        ; retur hvis indholdet ej er 0
     sta $dd0e      ; start tællerA
     lda.$55        ; hent startværdi for tællerB
     sta $dd0f      ; start tællerB
mk2  ldy $a2        ; hent lille byte jiffies
     cpy #60        ; retur hvis ej decimal 60
     bne mk2        ;
     stx $dd0e      ; stop tællerA
     stx $dd0f      ; stor tællerB
```

- o - 0 - o -

Valgopgave: Beregning af fart og acceleration.

Her skal tællerne bruges til at registrere den tid, det tager for vores luftpudevogn at flytte sig 10 cm. Herefter kan farten beregnes. I praksis vil vi med luftpudevognens fane bryde en lysbro; - og det kan datamaten registrere.
Impulserne til tælling på CNT2 kan hentes direkte fra en fototransistor, MEL 31. Se nærmere herom på side $08.
Når opstillingen er klar, kan du efterprøve følsomheden ved at peeke værdien på portB. - Husk: Den skal stilles som indgang og alle åbne bits giver "1", er høje. Fra programmet "jiffies" ved du, at phi-2 klokken går 985309 skridt frem på 1 sek. Fototransistoren tilsluttes bit 0 med collektor til bit 0 og emitter til stel = 0 volt. Når programmet senere skal beregne farten, vil vi arbejde

med en nøjagtighed på 1/10000 sek. Med denne måleenhed er vi over den fart COMAL kan klare.
Opgaven kan skematisk se sådan ud:



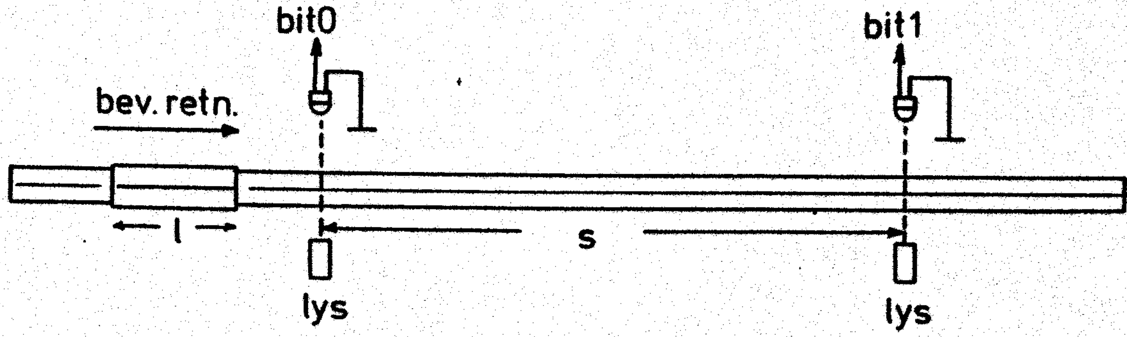
Herefter kan registrene $dd04 til $dd07 peekes.
Et forslag til assemblertekst findes som "src.tæller2" og det tilhørende COMAL program som "fartmåling1".

Acceleration.

I fortsættelse heraf kan også acceleration måles med to lysbroer. Ved acceleration menes fartændring pr. tidsenhed. - Der skal bestemmes 2 farter og den mellemliggende tid. De to lysbroer sluttes til bitt 0 og til bit 1. Med den valgte beregning, må bit 0 være den første lysbro, der passeres. Når opstillingen er på plads, kan du peeke værdierne under passage.af luftpudevognen. Er det rigtigt, at følgende skema gælder:

bit 0	bit 1	indhold i $dd01
lys	lys	$fc
mørk	lys	$fd
lys	mørk	$fe

Se tegningen herunder:

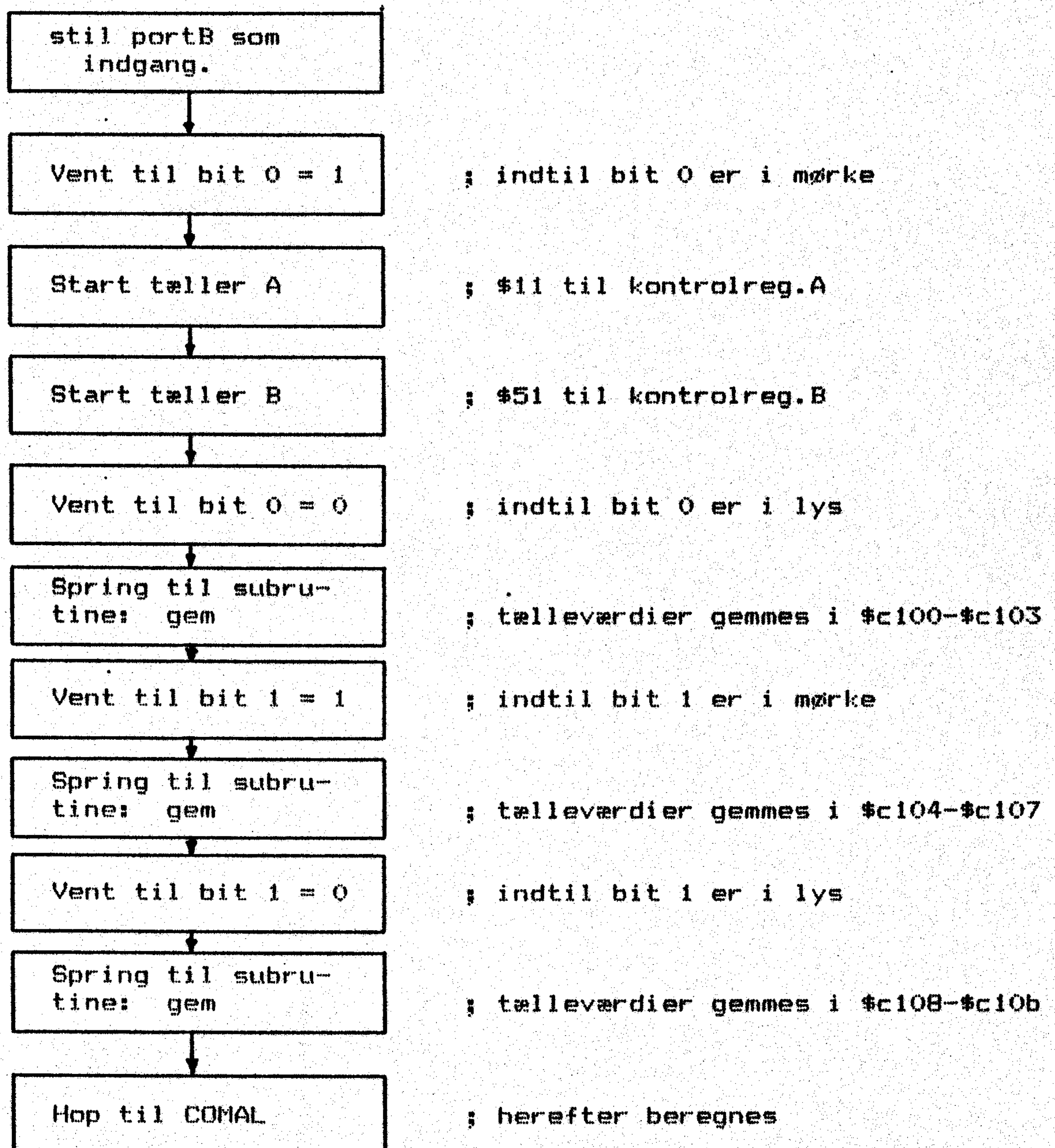


Vi skal bestemme:

tiden for passage af lysbro, bit 0 = tid0, fart0:=1/tid0
tiden for passage af lysbro, bit 1 = tid1, fart1:=1/tid1
tiden for passage fra bit 0 til bit 1 = acctid
acceleration:=(fart1 - fart0)/acctid.

De enkelte tidsaflæsninger må gemmes i lageret på en sådan måde, at de kan hentes igen via COMAL. Hertil bruges lageret fra $c100 og frem. Hver måling kræver 4 bytes.

Algoritmen kan være:



Eksemplerne "src.tæller1" og "acceleration1" kan vise dig på vej.

Vi har allerede brugt indexregistre, når vi fra en adresse ville tælle os frem til en ønsket lagerplads. Eksempelvis side 27-28, side 59 og side 67 i arbejdsbogen.
Her vil vi præsentere og gennemgå adresseringsmåden som et hele. Den er meget anvendt, når en sløjfe skal udføres med forskellig indlæsning for hver omgang. Den fylder meget lidt i lageret og er dermed hurtig.
Hvis du ønsker at arbejde med morsegeneratoren som valgopgave, vil det lette forståelsen, hvis du gennemarbejder dette afsnit grundigt.

Absolut x-indexeret adressering og
absolut y-indexeret adressering virker ens. Derfor gennemgås kun et eksempel.

Skrivemåden i assembler er: adresse,x - og udtrykket skal læses: Den absolutte adresse + indholdet i .x-reg giver slutadressen. Det er microprocessoren, der foretager denne beregning.

Som eksempel på denne adresseringsmåde vil vi løse en opgave, der går ud på at fylde de øverste 5 skærmlinjer med "snabel a". 5 linjer a 40 pladser er 200 pladser; adressen for første plads på skærmen er $0400 og den tilhørende farvehukommelse findes i adresse $d800 (se håndbog side 382/383). Om skærm- og farvehukommelse kan du også læse i arbejdsbogen side 15-16.

I assembler kan opgaven se sådan ud:

```
        .base $c000           ; programstart
        .global screen=$0400  ; label for skærmstart
        .global farve=$d800   ; label for farvehuk.
        .global clear=$e544   ; label for underrutine slet
        ;
        jsr clear             ; renser skærmen
        ldx #$00              ; læg 0 umiddelbart i .x-reg
 hop    lda #$00              ; læg skærmkode @ i .a-reg
        sta screen,x          ; gem indh. i .a-reg i celle $0400+x
        ; 1 - her indsættes senere en linje
        lda #$07              ; farvekode for gul
        sta farve,x           ; gem farven i celle $d800+x
        ; 2 - her indsættes senere en linje
        inx                   ; gør indh. i .x-reg 1 større
        cpx #$c8              ; sammenlign indh.i .x-reg med 200
        bne hop               ; indh. ej 200, så spring til hop
        ;
 tast   lda $91               ; læg indh. i celle 145 i .a-reg
        cmp #$7f              ; er værdien 127 (stop) ?
        bne tast              ; hvis ej - spørg igen
        ;
        rts                   ; retur til comal
        .end                  ; sluttegn for assembler c-64
```

Prøv om du kan assemblere dette program pr. håndkraft og skrive det ind som datalinjer i ladeprogrammet.
På disketten findes programmet som "src.skærm3" klar til behandling med assemblerprogrammet. Hvis du foretrækker at klare opgaven på den måde, skal assembler c-64 hentes ind og startes først.

Læg mærke til forklaringerne i de enkelte linjer. De letter forståelsen, så programmet også kan læses om 2 måneder, når du har

glemt alt om denne opgave. Når assembleren behandler programmet, vil den ikke læse det der står efter semikolon. Efter at label'erne er defineret, skal skærmen slettes. I Kernal findes en rutine, der starter i adresse $e544. Når den kaldes, slettes skærmen og programmet fortsætter. Så lægges 0 i .x-reg som tælletal og koden for snabel a i .a-reg. Indholdet i .a-reg gemmes i adresse 1024+indholdet i .x-reg, altså 1024+0, skærmens første plads. Koden for blå farve lægges i .a-reg og gemmes i farvehukommelsens første plads. Indholdet i .x-reg opskrives med 1 og det undersøges, om indholdet er er lig 200; - hvis ikke, hoppes til mærket "hop"; koden for snabel a lægges i .a-reg og gemmes i adresse 1024+indhold i .x-reg, der nu bliver 1025; gul farve lægges i den tilhørende farvehukommelse, indholdet i .x opskrives med 1 og det undersøges, om værdien har nået 200,......... - sådan gentages forløbet 200 gange før sløjfen forlades og programmet går videre til næste sløjfe, hvor det undersøges, om der er trykket på STOP. Til sidst springes tilbage til COMAL. De linjer, der indledes med et punktum, er pseudokoder, der skal bruges af assembler c-64.

I dette eksempel er forklaringen lavet meget fyldig. Det bliver ikke tilfældet med de næste eksempler. Derved kan de specielle nye ting bedre fremhæves.
Har du forøvrigt lagt mærke til, at det almindelige forløb i 6510 er, at en værdi hentes ind i .a-reg, behandles sammen med en værdi fra .x- eller .y-reg og lægges tilbage til en adresse i lageret. I eks. bruges indholdet i .x-reg til at lave en tællesløjfe med. For hvert nyt tællertal, flyttes til den tilsvarende celle i skærmhukommelsen og tegnet udskrives. Det er denne opgave, der hedder x-indexeret adressering. 6510 arbejder med en fast adresse og lægger hertil indholdet i .x-reg. En anden ting er brugen af labels (mærkeseddel) I første del bruges en label som en vejviser til en bestemt adresse. Det gør programmet lettere at læse. Senere bruges en label som et mærke, hvortil programtælleren skal springe, hvis bestemte betingelser er opfyldt. Skal en label bruges som et hopmærke, stilles den foran de mnemoniske koder. Hvis du bruger assembler c-64, behøver du ikke at tænke på de adresser, koderne skal lægges i. Når starten er angivet, her $c000, klarer programmet resten.

Kan du ikke huske betjeningen af assembler c-64, må du genopfriske vejledningen. Hent assembler c-64 ind i datamaten, start programmet med SYS 64738 og vælg A for indskrivning (A 100,10)<R>. Skriv de enkelte linjer, som de er anført. Husk at starte med de gennemgåede linjer først og slut det hele med .end. Se eksemplet her ved siden af. Gem kildeteksten på en diskette under navnet "src.navn"; og så kan selve assembleringen begynde. Når udlistningen er færdig, ligger der et stykke maskinkode på disketten som "obj.navn" og samtidig har du en udlistning på papir med koden i hexkode. Den er lige klar til at blive behandlet i programmet lav'data'linjer. Gennemfør opgaven og få programmet til at virke ordentligt.

HUSK: konstruktionen screen,x betyder: læg indholdet i .x-reg til screenadressen; så har du den aktuelle adresse. Hvorfor bruger man ikke et + ?
Dette tegn har en anden betydning, når assembleren skal behandle orden.

```
LDA screen,x beregnes af 6510 i maskinkode
LDA screen+1 beregnes af assembleren og kodes som en abs.adresse
```

Hvad sker der i programeksemplet, hvis du indføjer disse linjer på de angivne pladser:

```
 1  sta screen+280,x
 2  sta farve+280,x
```

- o - 0 - o -

Om relative spring.

Ved label hop, ses et relativt spring, der udføres hvis linjen bne hop er opfyldt.

BNE tydes: branch not equal zero; altså spring hvis indholdet i .x-reg er forskellig fra den angivne værdi - forskellen dermed ej 0.

I COMAL kunne det skrives: if (.x-200)<>0 then goto hop:

Ved et relativt spring menes, at springet angives fremad/baglæns som et antal trin i forhold til programtællerens nuværende plads. I stedet for at angive den absolutte adresse, oplyses det antal byte, der ligger mellem programtælleren og springmålet.

```
hop   lda #$00
        .
        .
        .
      bne $f1
PC -> lda
```

Hvis vi bruger eksemplet, står PC på lda, når bne hop er læst. Tælles byte bagud til lda efter label hop, vil du tælle 15 byte.

256-15 = 241 eller i hex $f1

Som omtalt under 2'er komplementtal, opfattes de første tal 0-127 som positive, tallene fra 128-255 som negative, idet bit 7 her er sat. 241 er altså et negativt tal med værdi -15. Du kan kontrollere ved at tælle på udlistningen. Denne lidt snurrige måde at beregne spring på skyldes bl.a., at der intet register findes, hvor man evt. kunne lægge oplysning om fortegn.

Måske skulle du lige repetere stykket om regning med binære tal, etkomplement og tokomplement - side 7-8.

- - o - - 0 - - o - -

Det var absolut indexeret adressering.
Vælger du at give dig i kast med valgopgaven "tekstbehandler", vil du her få præsenteret en anden indexeret adresseringsform, nemlig den indirekte, - men den tid - den glæde.

Valgopgave: Morsetræner.

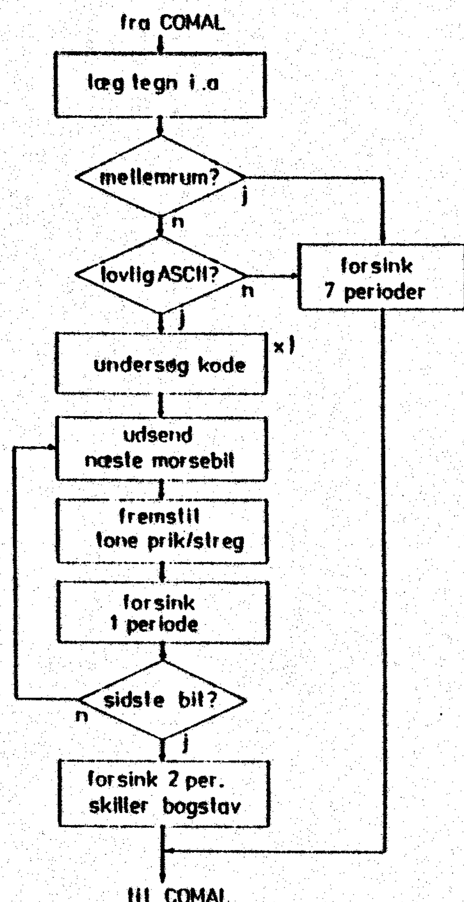
I nogle erhverv og hos radioamatører stilles krav om kendskab til morsealfabetet - derfor må det trænes. Her vil vi fremstille et program, der får datamaten til at omsætte indtastning af et bogstav til det tilsvarende morsetegn; der skal lyde i højttaleren og samtidig skal signalet sendes ud på userporten, hvor det kan ses på prøvepladen. Dette skal forestille, at datamaten styrer f.eks. en radiosender til telegrafi.
Det er en opgave med flere led, så den må deles op.

Først det overordnede om grundide og nødvendig lagerplads:
Der startes i COMAL med vejledning på skærmen, indlæsning af data og opstilling af startværdier. Meldingen skal indtastes og overgives til maskinkoden, der udsender morsekoden. Selve maskinkoden lægges fra $c000 og følges af en tabel med morsetegn. Vi får brug for et sted at lægge det aktuelle tegn samt plads for 3 variabler, idet farten skal kunne varieres.

```
zero page:                              $c000:

$55   kasse(akt.bogstav)               maskinkode
$fb   fart
$fc   count(int.tæller)                herefter
$fd   tegn(udsendelse)                 tabel med morsetegn
```

Tegnfarten kan ændres ved at ændre indholdet i $fb. Den valgte enhed er den tid, det tager at udsende en prik. Vi bruger phi-2 klokken, der arbejder med ca. 1 MHz. Hvis vi lægger decimal 100 i $fb, giver det en punktlængde på ca o,5 sek. Count og tegn bruges af maskinprogrammet selv. Rutinen kaldes når der ligger en lovlig værdi i fart og der står et tegn i kasse.



Hvordan kan et morsetegn udsendes ? Se blokdiagrammet her. Der hentes et tegn fra .a. Er det mellemrum ? - Hvis ja, indskydes 7 perioder vente. Så undersøges, om der står et lovligt ASCII-tegn i .a. Værdien skal ligge mellem $28 og $5d. (Se håndbogen) Ved ulovligt tegn, forsinkes afviklingen og der springes tilbage. Det lovlige tegn skal omsættes til morsekode, der hentes fra tabellen som en binær kode. (8 bit) Denne teknik vil jeg gøre rede for om lidt. Den tilhørende prik eller streg udsendes og der testes, om sidste bit er nået. Endelig springes tilbage for at hente næste tegn.

Morsetegnene må findes i en tabel eller i en håndbog. De skal omsættes til en binær kode efter følgende ide:

Start med en streng med 8 nuller. Hvert morsetegn skal kodes, så en prik er "0" og en streg er "1". Foran koden sættes et startbit:="1" og hele koden rykkes til højre i strengen. De ikke benyttede bit sættes til "0". Eks.:

ASCII	bogstav	morse	kode	hex
42	b	-...	%00011000	$18
56	v	...-	%00010001	$11
4b	k	-.-	%00001101	$0d
54	t	-	%00000011	$03
45	e	.	%00000010	$02
ikke kendt morsetegn			%00000001	$01

Når disse koder kaldes, må en byte undersøges fra venstre; bit for bit. Det første "1" er kodens start. De resterende bit indeholder morsetegnet. Når 8 bit er prøvet, er byten undersøgt. Dette holdes der styr på med tælleren count. For hver "1" skal der udsendes en streg, for hvert "0" en prik.
Det var måske en opgave for dig at lave et program i COMAL, der kunne omsætte ASCII-tegn til kode og beregne den tilsvarende hexværdi, når morsekoden indtastes. Koden skulle udskrives på papir. Programmet "tegnomsætter" er brugt til at fremstille den tabel, der findes på næste side. Opgaven er at ændre en ASCII-kode til den tilsvarende binære kode. Som eksempel tager vi bogstavet "b", morse "-...". Hver streg skal kodes som "1", hver prik som "0". Binærkoden er altså 1000. Ifølge forudsætningerne skal denne kode indledes med et startbit og der skal fyldes op til 8 bit med nul ler. Koden må derfor være b:=%00011000 eller $18. Det er denne kode, der undersøges i blokdiagrammet med x). Når alle koder er udregnet, skal tabellen indskrives lige efter maskinkoden med label: tabel. Da første tegn har ASCII $28 og vi vil bruge indexeret adressering for at finde rundt, må tegn kunne findes i tabellen med operanden: "tabel-$27,x.".

Fremstilling af tonen.
Her vil vi bruge tonegenerator 1. Se nærmere i håndbogen om styring af SID chip. På disketten ligger 2 eksempler som "csv.prøvelyd" Hent dem ind i datamaten og afgør om et af dem passer dig.

```
POKE $d400,$df // freq.lille byte
POKE $d401,$1c // freq.store byte
POKE $d405,%00010001 // attack/decay     store/lille nibbel
POKE $d406,%11110010 // sustain/release    -       -
POKE $d418,%00001111 // styrke                     -
POKE $d404,%00010001 // start trekant    form /start
for a:=1 to 500 do null
POKE $d404,%00010000 // stop trekant     form /stop
```

De første 5 linjer indstiller generatoren, i $d404 startes og stoppes og tælleren skal bare give tonen varighed. Den falder bort i det endelige program. Det er kun start- og stopværdi, der skal styres i maskinkoden, - de andre værdier pokes på plads allerede i COMAL-programmet.

Nu har vi skrevet tabellen og fremstillet tonen og kan kaste os over selve programmet. Find på disketten "src.morsetræner". Hent det ind med assembler c-64 og list kildeteksten ud på papir. Det er nemmest at følge med i gennemgangen, hvis du har den udlistede tekst ved siden af dig.

decimal	hex	tegn	morsekode	binærkode	hexkode
39	$27	'	.----.	%01011110	$5e
40	$28	(	-.--.-	%01101101	$6d
41	$29	)	-.--.-	%01101101	$6d
42	$2a	*		%00000001	$01
43	$2b	+		%00000001	$01
44	$2c	,	--..--	%01110011	$73
45	$2d	-	-....-	%01100001	$61
46	$2e	.	.-.-.-	%01010101	$55
47	$2f	/	-..-.	%00110010	$32
48	$30	0	-----	%00111111	$3f
49	$31	1	.----	%00101111	$2f
50	$32	2	..---	%00100111	$27
51	$33	3	...--	%00100011	$23
52	$34	4	....-	%00100001	$21
53	$35	5	.....	%00100000	$20
54	$36	6	-....	%00110000	$30
55	$37	7	--...	%00111000	$38
56	$38	8	---..	%00111100	$3c
57	$39	9	----.	%00111110	$3e
58	$3a	:	---...	%01111000	$78
59	$3b	;		%00000001	$01
60	$3c	<		%00000001	$01
61	$3d	=		%00000001	$01
62	$3e	>		%00000001	$01
63	$3f	?	..--..	%01001100	$4c
64	$40	@		%00000001	$01
65	$41	a	.-	%00000101	$05
66	$42	b	-...	%00011000	$18
67	$43	c	-.-.	%00011010	$1a
68	$44	d	-..	%00001100	$0c
69	$45	e	.	%00000010	$02
70	$46	f	..-.	%00010010	$12
71	$47	g	--.	%00001110	$0e
72	$48	h	....	%00010000	$10
73	$49	i	..	%00000100	$04
74	$4a	j	.---	%00010111	$17
75	$4b	k	-.-	%00001101	$0d
76	$4c	l	.-..	%00010100	$14
77	$4d	m	--	%00000111	$07
78	$4e	n	-.	%00000110	$06
79	$4f	o	---	%00001111	$0f
80	$50	p	.--.	%00010110	$16
81	$51	q	--.-	%00011101	$1d
82	$52	r	.-.	%00001010	$0a
83	$53	s	...	%00001000	$08
84	$54	t	-	%00000011	$03
85	$55	u	..-	%00001001	$09
86	$56	v	...-	%00010001	$11
87	$57	w	.--	%00001011	$0b
88	$58	x	-..-	%00011001	$19
89	$59	y	-.--	%00011011	$1b
90	$5a	z	--..	%00011100	$1c
91	$5b	æ	.-.-	%00010101	$15
92	$5c	ø	---.	%00011110	$1e
93	$5d	å	.--.-	%00101101	$2d

I opgaven anvendes x-indexeret adressering, idet koden skal findes i en tabel; - sløjfer indeni hinanden, der bestemmer forsinkelse; - styring af userport, idet en telegrafisender skal aktiveres, samt anvendelse af tonegenerator.

Hovedprogram for morsegenerator.

Før kald af maskinkoden, må sendehastigheden være bestemt og der må stå et bogstav i kasse. Da sløjferne er opbygget, så de giver et tidsforbrug på ca. 5 ms, skal en variabel i adresse $fb have en værdi omkring 100 for at give en punktlængde på 1/2 sek. Fra COMAL kan andre værdier lægges her og dermed påvirke tegnfarten.
Først lægges tegnet i .a. Er det mellemrum ? - spring til space. Ellers må det undersøges, om tegnet er lovligt. Værdier under eller over tegn og alfabet afvises, - her springes til exit. I øjeblikket sker der ikke mere her, men du kan senere udvide med en rutine, der udskriver en fejlmelding om, at der er indtastet et ulovligt tegn.
Er tegnet godkendt, må næste trin være at finde den tilhørende kode i tabellen, der ligger efter label: tabel. 1. byte i tabellen svarer til tegnet med ASCII $28. Når vi vil bruge x-indexeret adressering til at finde koden, må ASCII-tegnet skubbes til .x som en tæller. Derpå trækkes $27 fra kodestart og indholdet i .x tæller frem til korrekt tegn, Snedigt, hva' ?
Nu ligger "morsekoden" i .a. Husk, at den egentlige kode starter med "1" som startbit. Herefter følger nuller og enere, der danner selve koden. Indholdet i .a skubbes så mange gange til venstre, at startbit'et findes. De efterfølgende bit skal bruges til fremstilling af prikker og streger. Se programmet fra "startbit" og frem. Normalt ville man bruge .y som tæller for undersøgelsen, men rutinen "send", der fremstiller tonen bruger .y til tonens varighed. Så må .x kunne bruges, - men det går heller ikke. Rutinen "vent" skal bruge .x. Når vi ikke kan bruge tælleregistrene, må vi vælge en lagerplads på side 0 som tæller. Det er her count kommer ind. Selvfølgelig kunne vi gemme på stakken og hale frem igen senere, men oversigten i programmet ville dale.
Altså: De venstrestillede nuller i .a undertrykkes ved at indholdet i .a skubbes så mange gange til venstre, at startbit findes. Når dette er sket, vil hvert efterfølgende bit fremstille en prik eller en streg. Så snart det udskubbede bit er testet, springer vi til rutinen "send", hvor en tilsvarende kort eller lang tone fremstilles. Fordi indholdet i .a gennem venstreforskydningen ændres, må vi gemme det før hvert spring til "send". Det klares med instruktionen "sta tegn". Når den nye kode er sikret i tegn, lægges en værdi i .y som bestemmer tonens længde. $01 for prik, $03 for streg.
Måske skulle du repetere om bitoperationer, side 41. Ved forskydning til venstre, overføres bit nr. 7 til carry. Er det 0, er carry clear; er det 1 er carry set. I vores eksempel betyder det: Når bit 7 flyttes til carry, må vi teste med bcc(branch on carry clear)=bit er 0; svarende til en prik; værdien $01 bruges i rutinen send, - ellers bruges $03.
Send aktiverer tælleren med den valgte frekvens, der styres af phi-2 klokken. Vi har forberedt tonen, valgt kurveform og frekvens - nu skal den startes. Startværdien $11 lægges i $d404. Da vi samtidig vil kunne sende lysblink, må portb også tænde bit0. Tonens længde styres via underprogrammet "vent". Når det er udført, slukkes tonen og porten afbrydes. - Endelig må der indskydes en periode inden næste tegn. - Derefter må bittælleren decrementeres i count og der skal testes, om der er flere bit, der skal skydes gennem .a. - Hvis vi ikke er færdige, springes til next og forløbet gentages. Når bogstavet er udsendt, skal der indskydes 3 perioder inden næste tegn. Denne værdi kan ændres efter ønske.

Underprogrammet "vent" laver en forsinkelse på:

t:=(indhold i .y*4)*(fart)*5 ms

Den ønskede forsinkelse bliver fremstillet således:

```
vent    tya
        asl       ; a=a*2      De første 4 linjer ganger
        asl       ; a=a*2      indholdet i .y med 4. Det
        tay       ; y=y*4      sker med 2 gange asl. Et
mk3     lda fart               overløb kan ikke finde sted,
mk2     ldx #$fa               idet 4*7<255.
mk1     dex
        bne mk1
        sec                    subtraktionen må ske på den
        sbc #$01               viste måde, dels for at bruge
        bne mk2                tid, dels fordi .x og .y er
        dey                    optaget, dels for at vise .a
        bne mk3                brugt som tælleregister.
        rts
```

Sløjfen gennemløbes så mange gange, værdien i fart tillader.
Sålænge du ikke er fortrolig med morsekoden, bør du vælge en langsom afvikling. Værdierne kan evt. ændres yderligere i COMAL-programmet.

Nu kan kildeteksten assembleres og den fremkomne objektkode laves om til datalinjer. - Så mangler vi Comal-programmet, der skal holde sammen på det hele. Dette program, skal lægge maskinkoden på plads, give nogle vejledninger, indstille fart, musik og retningsregister.
Indstastning af melding kan klares i en sløjfe med LOOP, REPEAT, eller hvorden du foretrækker det.
Prøv med enkelttegn og undersøg dernæst, om der også kan skrives hele sætninger i sammenhæng.
Et forslag til indlæsning findes som "morsetekst.prøv" og et endeligt program som "morsegenerator".
Forsøg at lave din egen løsning først. Du får ikke maximalt udbytte af opgaven, hvis du går direkte til løsningen og bare konstaterer, at det fungerer ! !

- o - O - o -

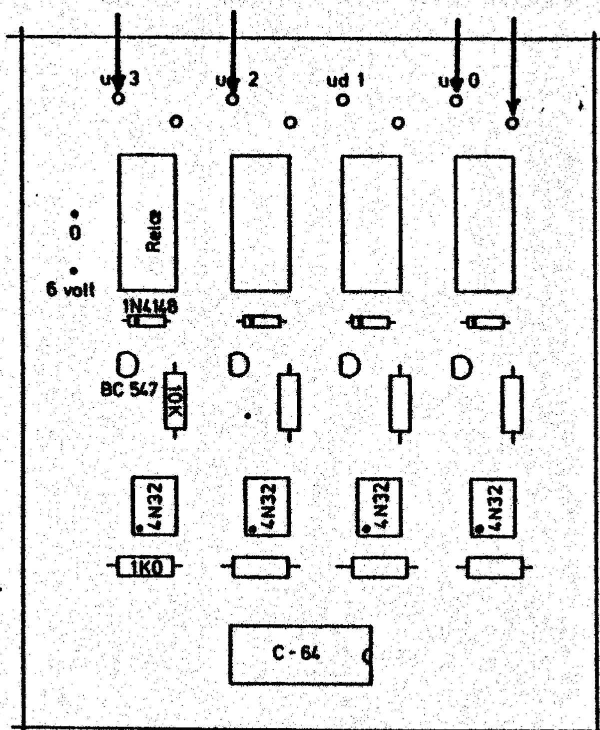
Ekstraopgaver:

a) Omskriv kildeteksten, så der på userporten vises ASCII-værdien af det udsendte tegn i stedet for blink i morsekoden.
Assembler, lav datalinjer og merge det rettede program til COMAL-programmet og få det til at fungere.

b) Udbyg kildeteksten, så der ved ulovlig indtastning lyder en tone og på skærmen fremkommer teksten: "ULOVLIG INDTASTNING !"
Meldingen skal stå på skærmen i nogle sekunder. Derefter fortsættes programafviklingen.
Fejlmeldingen kan stå som en tabel efter morsekoderne.

Styring af motorer.

Du kan læse om den praktiske opbygning af printplader med relækontakter på side $09 til $0c i de "tekniske sider". Her ved siden af ser du en skitse, hvor ledningsforbindelsen vises.



Forbindelsen til datamaten sker gennem det sædvanlige kabel. Skal du kunne stoppe og starte en motor, forbindes motoren mellem ud( ) og den tilhørende 0 (eksemplet til højre) - skal motoren styres venstre/højre om + start/stop, - må motoren forbindes til 2 ud( )-bøsninger(eksemplet til venstre).
I denne konstruktion er der for yderligere sikring af datamaten indskudt en optokobler i hver ledning, så der ingen elektrisk forbindelse er mellem datamat og ydre strømforsyning.

Opgaver: Se diagrammet side $09 forneden.

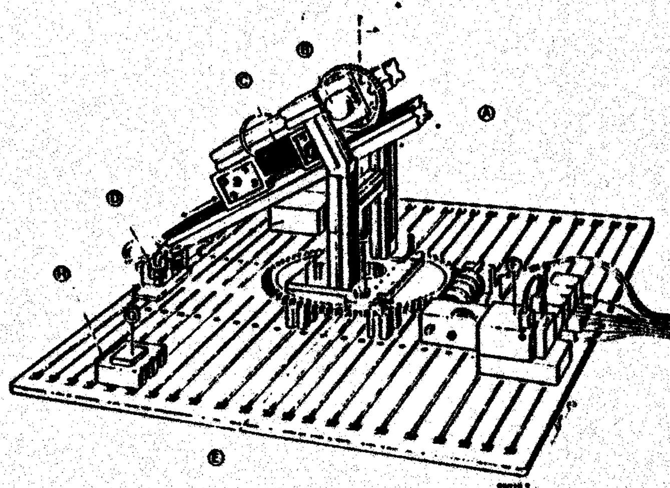
a) Hvad sker der med motoren, hvis bit1 går høj og bit0 stadig er lav ?

b) Hvad sker der med motoren, hvis bit0 går høj (bit1 er stadig høj) ?

Er følgende postulat sandt:

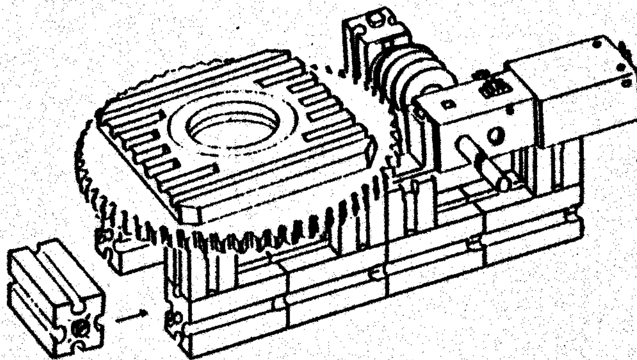
Ingen spændingsforskel mellem motorens poler = stilstand.

c) Skriv et lille program, der kan starte en motor, lade den køre i 10 sek. og derefter stoppe den.



Nu vil det være rart, om en kontakt kan stilles uden hensyn til de andre kontakters stilling. Det må ske gennem afmaskning med de logiske BITAND og BITOR. - Læs side 38-40 inden du går videre med opgaverne.

d) Opbyg en drejeskive af de udleverede klodser. Her ses den endelige form. På side $0d kan du finde en eksploderet skitse.



Du kan selvfølgelig skrive direkte til opstillingens motor, men en bedre løsning ville være, at fremstille et grundprogram, der indeholdt de procedurer der er nødvendige for at skrive et program. Se eksemplet på næste side. Det er et meget simpelt program, men det kan hjælpe dig over de første forhindringer.
Få drejeskiven til at fungere ved hjælp af programmet.

e) Sæt manden på drejeskiven og forbind motoren med styreenheden, så den kan dreje skiven både højre og venstre om.

f) Lav om på dit styreprogram, så du fra tasterne kan bestemme, om manden skal dreje den ene vej eller den anden vej rundt.

Du skal sikkert lave om i proceduren "PROC tastatur". Den valgte funktion skal stå på skærmen sålænge den udføres.

g) Lav en lampe ved siden af manden. Lampen skal stå på en grå klods og den skal lyse, når han ikke drejer.

I stedet for lampen kan laves en arm med en el-magnet fastgjort. Lav to borde af grå klodser og flade plader og overvej, hvordan du kan flytte en mønt (femøre) fra det ene bord til det andet.

Gennem disse øvelser har du vel indset, at du kan starte programmet for derefter at skrive enkeltkommandoer på skærmen. Hvis du har flere kommandoer (procedurekald) stående på skærmen samtidig, kan du med markørtasterne flytte på markøren og evt. gentage forløbet flere gange.

Et lille styreprogram.

En af COMALs stærke sider er brug af procedurer. Det fremgår forhåbentlig af eksemplet "robot11". Dine egne linjer kan føjes til dette program. Det er bare besværligt hele tiden at skulle fjerne egen linjer, når nye opgaver skal løses. Derfor hægter vi PROC hent'frem og PROC skjul'til(linje) på styreprogrammet.
COMAL kan behandle programmer med linjenummer op til 9999. PROC skjul'til(linje) gemmer styreprogrammet, så det ikke kan listes. Programlinjerne lægges op over 10000. Ønsker du at rette eller udvide dit program, kan det bringes frem med PROC hent'frem. Begge procedurer findes listet på diskette som "lst.hent" og "lst.gem".

```
0010 // save "@0:robot11"
0020 // * programmet kan starte opgaver
0030 // * i styring af motor m.v.
0040 // * version marts 87
0050 //
0060 DIM tegn$ OF 1
0070 //
0080 PROC init
0090   DIM to'potens(0:7)
0100   FOR a:=0 TO 7 DO to'potens(a):=2↑a
0110   dataret:=$dd03
0120   portb:=$dd01
0130   POKE dataret,%11111111
0140 ENDPROC init
0150 //
0160 PROC vent(sek)
0170   tid:=TIME
0180   WHILE TIME<=tid+sek*60 DO NULL
0190 ENDPROC vent
0200 //
0210 PROC set(udgang)
0220   POKE portb,PEEK(portb) BITOR to'potens(udgang)
0230 ENDPROC set
0240 //
0250 PROC reset(udgang)
0260   POKE portb,PEEK(portb) BITAND (255-to'potens(udgang))
0270 ENDPROC reset
0280 //
0290 PROC stoppe
0300   FOR a:=0 TO 7 DO
0310     reset(a)
0320   ENDFOR a
0330 ENDPROC stoppe
0340 //
0350 init
0360 PAGE
0370 PRINT AT 2,3: "* * klar ! * *"
0380 PRINT AT 4,3: "kommandoer: "
0390 PRINT AT 5,3: "frem = f, stop = s"
0400 PRINT AT 6,3: "afbryd = 0"
0410 tastatur
0420 //
0430 PROC tastatur
0440   REPEAT
0450     REPEAT
0460       tegn$:=KEY$
0470     UNTIL tegn$<>""
0480     t:=ORD(tegn$)
0490     IF t=70 THEN
0500       set(0) // frem
0510       PRINT AT 8,5: "Frem  "
0520     ELIF t=83 THEN
0530       reset(0) // stands
0540       PRINT AT 8,5: "Stands"
0550     ENDIF
0560   UNTIL t=48
0570   stoppe
0580 ENDPROC tastatur
0590
```

Du kan også se dem herunder:

```
PROC skjul'til(linje) CLOSED
  pt:=PEEK($16)+PEEK($17)*256
  LOOP
    lin:=PEEK(pt)*256+PEEK(pt+1)
    IF lin>9999 then lin:=-10000
    h:=PEEK(pt+2)
    EXIT WHEN lin>linje OR h=0
    POKE pt,lin DIV 256
    POKE pt+1,lin MOD 256
    PRINT "*",
    pt:+h
  ENDLOOP
ENDPROC skjul'til
//
PROC hent'frem CLOSED
  pt:=PEEK($16)+PEEK($17)*256
  lin:=10010
  LOOP
    h:=PEEK(pt+2)
    EXIT WHEN h=0
    POKE pt,lin DIV 256
    POKE pt+1,lin MOD 256
    PRINT "*",
    pt:+h; lin:+10
  ENDLOOP
ENDPROC hent'frem
```

Nu vil vi lave et lille styreprogram, der ikke skal slettes før hver omskrivning. Vi bestemmer os til, at "robot11" indtil linje 400 med skal danne grundlag.

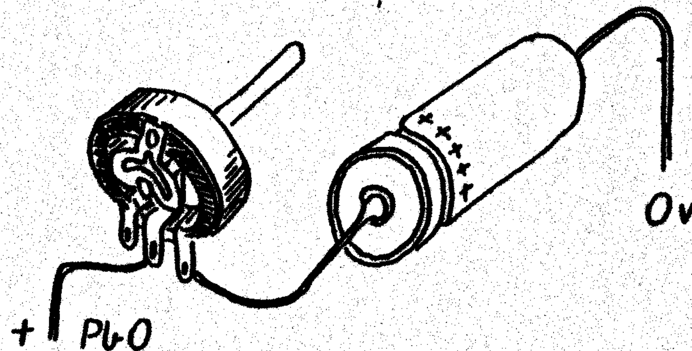
Hent "robot11" ind i datamaten. DEL 410- <R>; - MERGE "lst.hent" <R>; MERGE "lst.gem" <R>. Nu hænger programmet sammen. Find sidste linjenummer og husk det. Tast <f-7> og programmet startes. Skriv dernæst skjul'til(linje) <R> og en række stjerner på skærmen fortæller, at programmet gemmes.
Kan du liste programmet ? Skriv EDIT <R>, kan det lade sig gøre. Hvilken linje kommer frem, når du skriver AUTO <R> ? - Styreprogrammet er gemt, - kun dit eget program kan listes. Dit program slettes med DEL 1- <R> og et nyt kan skrives. Hvis du bruger NEW <R>, slettes alt og styreprogrammet må hentes ind igen.

h) Hvori består forskellen mellem "robot11" og "robot12". Begge programmer ligger på disketten.

Har du lyst til at trænge dybere ind i styringsopgaver, kan du undersøge programmet "styrDOZER". I dette program findes egen editor, der muliggør at du kan skrive ordre på skærmen og få dem udført flere gange. Der kan slettes, rettes og tilføjes i en skreven række af kommandoer. Du kan bruge taster og Joystick til kørslen, og programmer kan hentes og gemmes via disketten. Den vejledning der findes i programmet, skulle være fyldig nok til at du kan få DOZER til at køre. DOZER er vores bæltekøretøj med frontskovl. Om opbygningen af dette, kan du læse i de tekniske sider bag i bogen. Når programmet startes, vises en pop.up-menu med kommandoer. Den kan kaldes frem senere ved at taste ESC(RUN/STOP). Når programmet er i en INPUT-sætning, kan menuen ikke kaldes frem.

Opbygning af analog/digital converter.

Nu har du prøvet at stoppe og starte en motor, men du må selv overvåge og gribe ind, hvis motoren drejer for langt. Det vil være ønskeligt, om datamaten selv kunne overvåge arbejdet. Det er muligt, men først vil vi undersøge kombinationen en variabel modstand (potmeter) i serie med en kondensator.



Se tegningen herover. Et potmeter på f.eks. 4K7 og en elektrolytkondensator på 1000 uF/25V forbindes til portb via den store koblingsplade til 0 volt og bit0 i portb. Vær omhyggelig med at forbinde kondensatoren korrekt.
Stilles portb som indgang, sættes alle bit høje (ca. 5 V) og kondensatoren fyldes igennem modstanden indtil der ligger en spænding nær 5 V over kondensatoren. Stilles portb som udgang, sættes alle bit lave (ca. 0.1 V) og kondensatoren aflades gennem modstanden. Spændingen går mod 0 V.
Vi indskyder en tællesløjfe, som kører i den tid, det tager for kondensatoren at slippe af med sin ladning. Tællerens sluttal skal herefter vises på skærmen.

```
0010 retreg:=$dd03
0020 portb:=$dd01
0030 //
0040 POKE retreg,1 // bit0 udgang
0050 POKE portb,0
0060 vent(.2)
0062 //
0070 måling
0080 //
0090 PROC måling
0100   tæller:=0
0110   POKE retreg,0
0120   WHILE NOT (PEEK(portb) BITAND 1)>0 DO
0130     tæller:+1
0140   ENDWHILE
0150   POKE retreg,1
0160   PRINT tæller
0170   vent(.5)
0180   måling
0190 ENDPROC måling
0200 //
0210 PROC vent(sek)
0220   tid:=TIME
0230   WHILE TIME<=tid+sek*60 DO NULL
0240 ENDPROC vent
```

Hvilken indvirkning har potmeterets stilling på dette tal ?

Saml opstillingen, evt. med minidiller, indtast programmet fra forrige side og start.
Når du drejer på potmeterets aksel kan du se, om din antagelse var rigtig.

Hvad betyder kondensatorens størrelse for tællertallet ?

Skift til en anden kondensator. Var dit bud rigtigt ?

På denne måde har vi en mulighed for at følge potentiometerets drejning og kontrollere dets stilling gennem det tællertal, der hele tiden udsendes.
Den stilling armen i potentiometeret har, bestemmer den modstand, der er indskudt. Der er analogi mellem disse værdier. Den eneste slags oplysning en datamat kan reagere på er digitale. Her ser vi et eksempel på at en analog oplysning (armens drejning) omsættes til en digital værdi (tællertallet). Denne konstruktion kaldes for en analog/digital omsætning, eller på udenøsk en A/D convertering.
Datamaten er født med 2 sæt A/D omsættere (altså 4 stk.). De er gemt i joystickportene. Kondensatorerne ligger i datamaten, mens vi kan tilslutte potmetrene udenfor. Samtidig kan de samme porte også reagere på elektriske kontakter. Strømskemaet kan ses på side $0e, hvor stikket ses udefra. Variablerne fra potx og poty er en talværdi mellem 0 og 255, der bestemmes af den spænding, der ligger mellem klemmen og + 5 V. Jo mindre modstand, jo lavere talværdi. Sammenhængen er ikke lineær og den afhænger også af den ind skudte kondensator. Kapaciteteen kan ændres med at indsætte en ydre kondensator ved C ext. Koblingspladen "J-port" giver mulighed for at arbejde med denne indgang. Variablerne tast'a og tast'b er 0 sålænge tasterne ikke er indtrykkede.
I COMAL-kapslen findes en pakke, paddles, der understøtter denne funktion. Den gøres kendt med USE paddles og rummer en procedure med kaldet: paddle(portnr,potx,poty,tast'a,tast'b). Alle værdier skal aflæses, men du behøver kun at bruge nogle af dem. På samme måde aktiveres joystick-pakken med USE joysticks. Her er procedurekaldet: joystick(portnr,retning,knap). Retning kan have værdierne:

```
              1                              frem
         8         2
    7         0         3           venstre    0     højre
         6         4
              5                              bak
```

Nu vil vi bygge videre på vores drejeskive med magnetarm. Se skitserne side $0e. Der skal indsættes et potentiometer på 4K7 ohm i drejeskivens faste del. Du klarer nemmest korrekt poling ved at føre + 5 V fra datamaten frem med rød ledning og bruge hunstik. Kombinationsstikket sættes i port2. I port1 skal sidde et joystick.

Opgaven består nu i at få magnetarmen til at hente en femøre fra et af de små borde og dreje den over til siden og aflevere den i en beholder. Du skal styre det hele med joystick og taster og de enkelte kommandoer lægges ind i en tabel, så datamaten kan huske kommandoerne og gentage dem et ønsket antal gange. Opgaven kunne udvides, så styreprogrammet blev gemt på diskette som en seq fil. Derved var det muligt at gentage opgaven en anden gang.

Vi kunne begynde med at klarlægge, hvilke vigtige variabler, vi burde have:

pos'sæt$(n) skulle rumme de enkelte styrekoder.
aktuel'pos kunne være armens nuværende stilling.
ønsket'pos den stilling armen skulle nå til.

Fra tastaturet kunne vi bruge "t", "s" og "v" for tag'mønt, slip'mønt og vent.
Brug det lille styreprogram "robot11" som udgangspunkt, lav lidt om på vejledningen på skærm, hvor armens stilling skal opdateres. Joystick skal bruge 3 for højre'drej, 7 for venstre'drej og FYR for markering af armens stilling.
Skal kommandoerne kunne gemmes, må de indsættes i en tabel, hvor de enkelte ordre lægges i nummerorden. Variablen kalder jeg pos'sæt(nr) og reserverer plads til 100 sæt. Der skal bruges en tæller som indeks (nr). En position fra paddle'a har indtil 3 cifre (max.255). For at alle positioner skal fylde lige meget, laves drej$ af en nulstreng "000"+STR$(aktuel'pos). Derpå skæres de bagerste 3 cifre ud af denne streng og tildeles pos'sæt$(nr).
List "robot21" ud på papir og følg programmets opbygning. Sæt dig sammen med din makker og forklar ham algoritmen (fremgangsmåden).
Her ligger et forslag, der kan bruges, hvis du selv kører fast, eller vil have nogle impulser. "robot22" giver mulighed for at gemme og hente styreprogrammet på diskette.

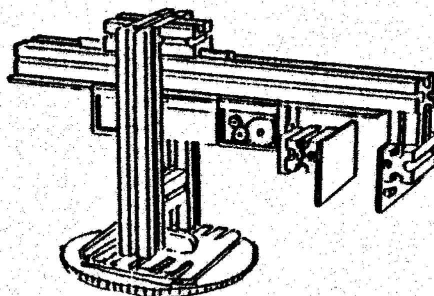
Som den sidste udbygning i denne afdeling, skal vi have fremstillet et transportbånd, der skal køre en bestemt tid, når der lægges en mønt fra magneten over på båndet. Se eksemplet "robot23".

- o - O - o -

Udbygning af robot 2X som ekstraopgave.

Se på tegningerne side $0f. Den faste arm kunne udbygges med en motordrevet glider, der kan fungere som gribeklo. Kloens løse kæbe kører på en tandstang, hvor dens position følges via et potentiometer. Nu er der fare for at kæben lukker helt og motoren kører videre, eller at kæben kører helt bagud af armen. Begge dele giver havari, så det må forhindres. Indbyg 2 mikrokontakter på den faste arm. Den ene kontakt skal aktiveres, når kæben er lukket; - den anden når kæben er maximalt åben. Disse kontakter sluttes til a'tast og b'tast. Du kan selv bestememe om kontakten skal ændre status ved tast=FALSE eller tast=TRUE, der er bøsning til begge løsninger.

Det er klogt at lave en forsøgsopstilling som antydet på øverste tegning og få den til at fungere korrekt, så løberen standser ved de to kontakter. Se "stoptast" på disketten. Kig endvidere på udlistningen næste side. Joystick styrer kørselen, paddles og tast returnerer de aktuelle positioner. Det ville være klogt at fremstille et diagnoseprogram, så du kan kontrollere kablerne inden du starter programmet.
Hele programmet findes som "robot24" på disketten.



```
1880 //
1890 kør'til'start
1900 TRAP ESC-
1910 nr:=1
1920 WHILE NOT ESC DO
1930   PRINT AT 16,25: "punkt nr. ",nr
1940   PRINT AT 22,1: USING "Nuværende position : ### ": aktuel'drej
1950   PRINT AT 23,1: USING "Nuværende arm      : ### ": aktuel'arm
1960   joystick(1,retning,knap)
1970   IF retning<>0 THEN
1980     CASE retning OF
1990     WHEN 1
2000       kør'frem
2010       LOOP
2020         paddle(2,a'paddle,b'paddle,a'knap,b'knap)
2030         joystick(1,retning,knap)
2040         EXIT WHEN retning<>1 OR a'knap
2050       ENDLOOP
2060       stop'kør
2070     WHEN 3
2080       drej'højre
2090       LOOP
2100         joystick(1,retning,knap)
2110         EXIT WHEN retning<>3
2120       ENDLOOP
2130       stop'drej
2140     WHEN 5
2150       kør'tilbage
2160       LOOP
2170         paddle(2,paddle'a,paddle'b,a'knap,b'knap)
2180         joystick(1,retning,knap)
2190         EXIT WHEN retning<>5 OR b'knap
2200       ENDLOOP
2210       stop'kør
2220     WHEN 7
2230       drej'venstre
2240       LOOP
2250         joystick(1,retning,knap)
2260         EXIT WHEN retning<>7
2270       ENDLOOP
2280       stop'drej
2290     OTHERWISE
2300       NULL
2310     ENDCASE
2320     IF a'knap OR b'knap THEN vent(2)
2330   ENDIF
2340   //
2350   IF knap THEN marker'punkt
2360   sv$:=KEY$
2370   CASE sv$ OF
2380   WHEN "V","v" // vent
2390     REPEAT
2400       INPUT AT 20,1: "Indtast ventetid i sek.(max.60) ": tid
2410     UNTIL 0<=tid AND tid<=60
2420     PRINT AT 20,1: SPC$(40),
2430     pos'sæt$(nr):="v"+STR$(tid)
2440     nr:+1
2450   OTHERWISE
2460     NULL
2470   ENDCASE
2480 ENDWHILE
2490 stoppe
2500 ENDPROC indlær
```

Universalprint og motorstyring.

I de tidligere eksempler skete tilbagemelding fra robotten via et potentiometer eller en kontakt, der benyttede paddle eller joystick. En mere simpel løsning er at programmere nogle bit i portb som indgange og slutte dem til trykkontakter. Se universalprintet side $10. Det rummer 4 indgange, sluttet til bit0 til bit3. - Men først en lille øvelse:

Tilslut dit print med lysdioder og skriv direkte til skærmen uden linjenumre:

```
poke $dd03,$ff <return>  - hvordan lyser LED ?
poke $dd03,$00 <return>  -    -    -    -
poke $dd03,$f0 <return)  -    -    -    -
```

Kan du forklare de 3 kombinationer ?

Måske går det bedre, hvis du skriver:

```
poke $dd03,%11110000 <return>
```

Store byte stilles som udgang, og der er ingen spænding her. Lille byte stilles som indgang, og der er 5 volt på kontakterne. Spørger du til portb med

```
print peek($dd01), får du svaret 15.
```

Alment må det om portb siges, at den starter med 0 volt (lav) på kontakterne, hvis de stilles som udgange; - og med 5 volt (høj) på kontakterne, hvis de stilles som indgange.

Når du får at vide, at bit0 til bit3 laves til indgange ved at skrive:

```
poke $dd03,$f0,
```

og at du som sædvanlig kan spørge til portens status med:

```
peek($dd01),
```

skal du lave et program i COMAL, der renser skærmen og viser indholdet i portb på skærmen, f.eks. fra 5.linje, 3.plads. En de luxe-udgave kunne have lidt tekst på skærmen, der forklarede det viste tal. Der er en funktion fra tidligere lektioner, der kan omsætte til binær notation. Forsyn universalprintet med 3 eller 4 trykkontakter, saml det hele og få programmet til at fungere. På skærmen skal du kunne se, hvilke kontakter, der er nedtrykkede.

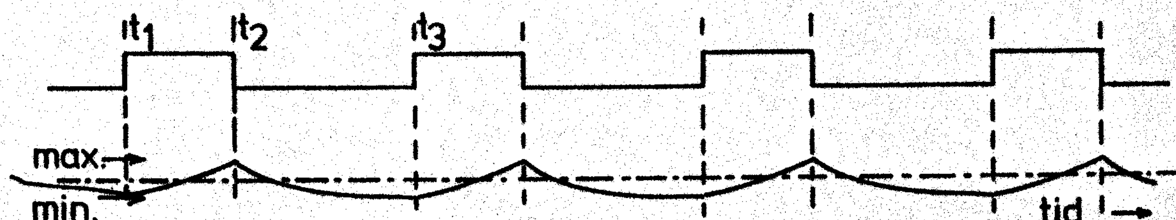
- o - O - o -

Samme print rummer også en motorudgang, der styres af transistorer. Næste opgave går ud på at fremstille et program, der starter motoren og får den til at køre højre om, hvis tast2 er trykket ind og venstre om, hvis tast0 er trykket ind - andre taster skal ignoreres.
Her skal både arbejdes med indgang og udgang på portb, så det er nok klogt at repetere stykket med binære operationer på side 38 - 40, inden du fortsætter.
Kan forslaget på næste side bruges, evt. i forbindelse med "robot11" ? - Se eksemplet på disketten og forslaget på næste side.

```
0010 // save "@0:motorstyr1"
0020 // * programmet kan starte opgaver
0030 // * i styring af motor m.v.
0040 // * version marts 87
0050 //
0060 DIM tegn$ OF 1
0070 //
0080 PROC init
0090   DIM to'potens(0:7)
0100   FOR a:=0 TO 7 DO to'potens(a):=2↑a
0110   dataret:=$dd03
0120   portb:=$dd01
0130   POKE dataret,%00110000
0140 ENDPROC init
0150 //
0160 PROC vent(sek)
0170   tid:=TIME
0180   WHILE TIME<=tid+sek*60 DO NULL
0190 ENDPROC vent
0200 //
0210 PROC set(udgang)
0220   POKE portb,PEEK(portb) BITOR to'potens(udgang)
0230 ENDPROC set
0240 //
0250 PROC reset(udgang)
0260   POKE portb,PEEK(portb) BITAND (255-to'potens(udgang))
0270 ENDPROC reset
0280 //
0290 PROC stoppe
0300   FOR a:=0 TO 7 DO
0310     reset(a)
0320   ENDFOR a
0330 ENDPROC stoppe
0340 //
0350 init
0360 PAGE
0370 LOOP
0380   port:=PEEK($dd01)
0390   afmask:=port BITAND %00001111
0400   IF (afmask BITAND %00000001)=1 THEN
0410     set(4)
0420     PRINT AT 5,4: "højre     "
0430   ELIF (afmask BITAND %00000100)=4 THEN
0440     set(5)
0450     PRINT AT 5,4: "venstre   "
0460   ELSE
0470     reset(4); reset(5)
0480     PRINT AT 5,4: "motorstop"
0490   ENDIF
0500 ENDLOOP
```

Fartregulering af motor.

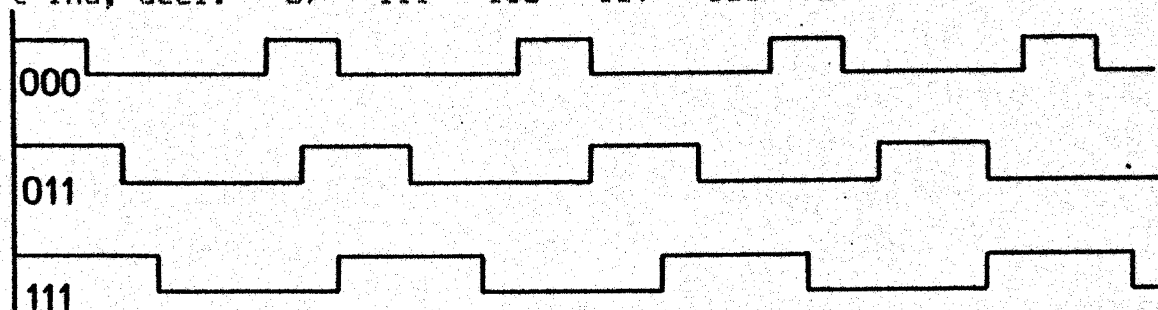
Hvis en motors omdrejningstal skal kunne ændres uden at drejningsmomentet ændres samtidig, kan det ikke klares ved at variere spændingen. En løsning vil være at hakke strømmen i stykker og sende den til motoren med et fast antal "klumper" pr. sek.
COMAL er for langsom til denne opgave; - den må løses i maskinkode. I eksemplet her vil vi benytte den vanlige jævnstrømsmotor (Fischer MINI-MOTOR) der styres med 3 tryktaster. Bitkombinationerne giver mulighed for 8 forskellige farter.
Først en ADVARSEL: Til den opgave, vi skal igang med, må ikke benyttes relæer. De kan overhovedet ikke følge med. Kun pladerne med transistorstyring eller universalpladen må bruges.



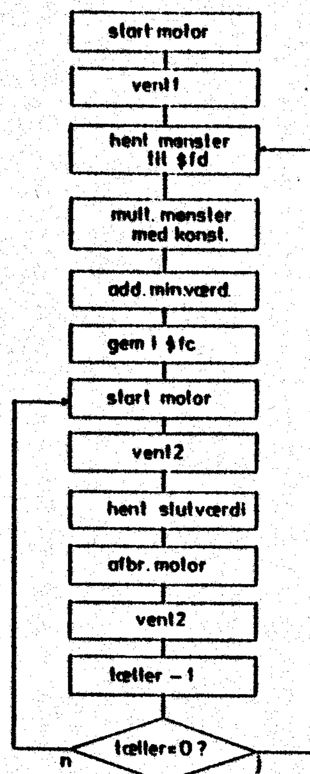
Ideen er at udnytte motorens træghed: Sende en firkantimpuls til motoren, hvorved omdrejningstallet stiger; - afbryde en tid, hvorved omdrejningstallet falder; - sende en ny firkant afsted inden motoren er gået i stå. Diagrammet viser, at motoren drejer langsomt ved t-1. - Så ankommer firkanten og drejningen vokser. Max.værdien nås ved t-2, hvor strømmen afbrydes og drejningen går ned. Ved t-3 ankommer næste firkantimpuls.
I praksis går det hele så hurtigt, at motoren indstiller sig på et omdrejningstal, der er antydet som middelværdien. Ved at ændre længden af firkanterne, er det muligt at indstille til forskellige omdrejningstal. I det viste forslag er der valgt nogle værdier, der kan være udgangspunkt for forsøg. Motoren er konstrueret til 6 volts drift. Da der ligger 2 diodestrækninger mellem stel og Vcc, må vi op på 7,5 volt driftspænding for at få fuldt udbytte.
Motoren skal kunne køre begge veje og derfor vælges universalprintet med bit0-bit2 som indtaster og bit4-bit5 som motorstyring.
Omdrejningsretningen vælges fra COMAL og lægges i $fe før kald af maskinkoden. Tastkombinationen hentes fra porten og lægges i $fd, mens $fb og $fc rummer tællere for venterutinerne. Tiden t-ind er den tid strømmen er indkoblet, t-ud den tid motoren er strømløs. Slukkettiden t-ud er valgt som en fast værdi $20. Impulstiden t-ind kan antage værdier, der er bestemt af bytemønsteret fra de indtrykkede taster, hvor %000 er laveste, %111 højeste værdi. Den må være sammensat af en mindsteværdi+konstant*bytemønsteret. Se tabellen herunder:

t-ud = $20 (fast værdi)
t-ind = $3f+$18*bytemønster

bytemønster	000	001	010	011	100	101	110	111
t-ind, deci.	87	111	135	159	183	207	231	255



Værdierne på foregående side er valgt som et udgangspunkt. Du kan selv forsøge dig med andre tal for t-ind og t-ud. Grafen giver dig sikkert en ide om funktionen.
Herunder kommer bemærkningerne til kildeteksten og blokdiagrammet viser ideen.



Når vi kører med motoren, indkobles den en tid for at opnå et passende omdrejningstal. Så indlæses den værdi, der ligger i kontakterne og den tilsvarende forsinkelse beregnes. Bitmønsteret multipliceres med en konstant, $18 og adderes til mindsteværdien $20. Resultatet heraf lagres i adresse $fc. Så indkobles motoren i den beregnede tid, vent2. Så bliver motoren afbrudt en tid. Denne løkke genenemløbes flere gange, hvorved omdrejningstallet stabiliseres, Så spørges om status for tasterne; - den kunne jo være ændret. Læg mærke til, at denne stadige forespørgsel betyder, at vi kan se bort fra kontaktprell. Subrutinen vent1 er hentet fra trafiklyset og vent2 er strikket sammen til formålet.

Hvis du har svært ved at fremskaffe 7,5 volt, kan du måske bruge en anden motor. En stor legetøjsfabrik i det midtjydske fremstiller en motor, der skal bruge 4,5 volt. Den passer til universalprintet og de 6 volt, vi har til rådighed, men en tekniker som du kan lave underværker med et stykke ledning, et element og en loddekolbe.

Herunder ser du indledningen til kildeteksten. Den fortsætter på de næste sider med det færdige COMAL-program. På disketten findes hele opgaven med vore aftalte navne:

src.motorreg	kildeteksten (brug assembler c-64)
obj.motorreg	objektkoden
dat.motorreg	datafilen (obj. via "lav'data'linjer")
csv.motorreg	COMAL save-fil

Forsøg dig nu med opgaven. God fornøjelse.

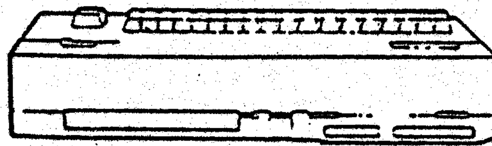
```
180;
190            .base $c000
200;
210; variabler defineres
220;
230            .global kasse1=$fb; tæller for vent1
240            .global kasse2=$fc; tæller for vent2
250            .global retn=$fe; omløbsretning
260            .global retreg=$dd03
270            .global portb=$dd01
280            .global get=$ffe4; undersøger tasttryk
290;
```

```
290;
300; Her begynder assemblerprogrammet
310;
320motor      lda #%11110000
330           sta retreg     ; bit4 til bit7 udgange
340           lda retn       ; omløbsretning hentes
350           sta portb      ; motor højre/venstre om
360           lda #$ff
370           sta kasse1     ; tæller for vent1
380           jsr vent1
390mstart     lda portb      ; hent indhold
400           and #$07       ; afmask 5 største bit
410           tay            ; kontaktværdi til .y
420           lda #$18       ; konstant ved skift
430           sta kasse2+1   ; gemmes i $fd
440add        cpy #$00
450           beq mk3
460           clc
470           adc kasse2+1   ; læg indh. $fd til sig selv
480           dey
490           jmp add
500mk3        sta kasse2+1   ; her er $18*tastværdi
510           lda #$3f
520           clc
530           adc kasse2+1
540           sta kasse2+1   ; her er $3f+$18*tastværdi
550           ldy #$80       ; antal gennemløb
560motind     lda kasse2+1
570           sta kasse2     ; tæller for vent2
580           lda portb
590           ora retn
600           sta portb      ; motor startes
610           jsr vent2
620           lda #$20       ; værdi forsink t-ud
630           sta kasse2
640           jsr vent2      ; tænder motor kort
650motud      lda portb
660           and #%00001111; afmask portb
670           sta portb      ; slukker motor
680           jsr vent2
690           dey
700           cpy #$00       ; skal gennemføres $c0 gange
710           bne motind
720           jsr get
730           cmp #83        ; er der trykket på "s"
740           beq ud
750           jmp mstart
760ud         rts
770vent1      ldx #$9d
780mk2        ldy #$71
790mk1        iny
800           cpy #$00
810           bne mk1
820           inx
830           cpx #$00
840           bne mk2
850           inc kasse1
860           lda kasse1
870           cmp #$00
880           bne vent1
890           rts
900vent2      ldx kasse2
910mk4        dex
920           cpx #$00
930           bne mk4
940           rts
950           .end
```

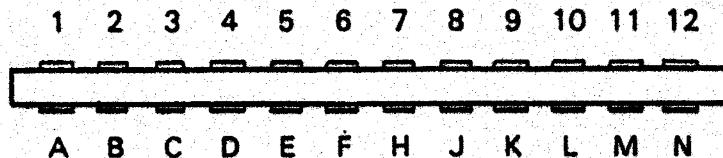
```
0010 // save "@0:csv.motorreg"
0020 // delete "motorreg"
0030 // * demonstrerer fartregulering
0040 // * via tryktaster - følg vejl.
0050 // * på skærmen.
0060 // * vers. jan.89
0070 //
0080 DIM sv$ OF 1
0090 indlæs
0100 PAGE
0110 PRINT AT 5,10: "Langeskov Ungdomsskole"
0120 PRINT AT 7,14: "Fartregulering"
0130 PRINT AT 10,3: "Monter 3 trykkontakter og forbind dem"
0140 PRINT AT 11,3: "med bit0-bit2 på universalprintet. -"
0150 PRINT AT 12,3: " - Motoren sluttes til bit4-bit5. "
0160 PRINT AT 13,3: " - Vælg om motoren skal dreje højre-"
0170 PRINT AT 14,3: "eller venstreom. Der stoppes med 's'."
0180 PRINT AT 16,3: "Vælg fart med tasternes bitmønster."
0190 REPEAT
0200   INPUT AT 18,3,1: "Højre/venstre (h/v) ": sv$
0210 UNTIL sv$ IN "HhVv"
0220 IF sv$ IN "Hh" THEN
0230   POKE $fe,%00100000
0240 ELSE
0250   POKE $fe,%00010000
0260 ENDIF
0270 //
0280 TRAP ESC-
0290 SYS $c000
0300 //
0310 PROC indlæs
0320   adr:=$c000
0330   LOOP
0340     READ byte#
0350     POKE adr,byte#
0360     adr:+1
0370     EXIT WHEN EOD
0380   ENDLOOP
0390 ENDPROC indlæs
0400 //
0410 //DATA $c000 // start adresse
0420 DATA $a9,$f0,$8d,$03,$dd,$a5,$fe,$8d
0430 DATA $01,$dd,$a9,$ff,$85,$fb,$20,$62
0440 DATA $c0,$ad,$01,$dd,$29,$07,$a8,$a9
0450 DATA $18,$85,$fd,$c0,$00,$f0,$07,$18
0460 DATA $65,$fd,$88,$4c,$1b,$c0,$85,$fd
0470 DATA $a9,$3f,$18,$65,$fd,$85,$fd,$a0
0480 DATA $80,$a5,$fd,$85,$fc,$ad,$01,$dd
0490 DATA $05,$fe,$8d,$01,$dd,$20,$79,$c0
0500 DATA $a9,$20,$85,$fc,$20,$79,$c0,$ad
0510 DATA $01,$dd,$29,$0f,$8d,$01,$dd,$20
0520 DATA $79,$c0,$88,$c0,$00,$d0,$da,$20
0530 DATA $e4,$ff,$c9,$53,$f0,$03,$4c,$11
0540 DATA $c0,$60,$a2,$9d,$a0,$71,$c8,$c0
0550 DATA $00,$d0,$fb,$e8,$e0,$00,$d0,$f4
0560 DATA $e6,$fb,$a5,$fb,$c9,$00,$d0,$ea
0570 DATA $60,$a6,$fc,$ca,$e0,$00,$d0,$fb
0580 DATA $60
0590 //DATA $48b4 // kontrolsum
0600 END "Genstart med <f-7> !"
```

Monteringsvejledning og teknisk beskrivelse.

Commodore C-64 har mulighed for at styre 8 kontakter gennem sin brugerport (I/O port). Brugerporten bag på datamaten ses ved pilen på denne skitse.



Betragtes den nærmere, er der på printpladen 12 printbaner på pladens overside (mærket 1-12) og 12 printbaner på pladens underside (mærket A-N). Det er ikke alle printbaner, vi kan bruge på dette tidspunkt, men den fuldstændige liste findes i prog.ref.guide side 359-360. En stor tegning af printbanerne ser du herunder. Bemærk at ben 1, 12, A og N alle er stel. Ben 2 kan levere 100 mA. Må kun bruges til lysdioder og små transistortrins basis. Ben C til L er port B's enkelte bit. Derforuden skal vi i forbindelse med tæller, reset m.v.



Holdes kantstikket i samme stilling som I/O stikket, ser du ind i 24 loddeøjne og de samme benævnelser er støbt ind i stikket. Det er dette billede der vises på tegningen med lakridsbændel og stik. Det er klogt at mærke kantstikket med "opad", - så undgår du fejlmontering. Der kan fås nogle kodestifter, der korrekt anbragt i stikket umuliggør fejlmontering.

Montering:

Begynd med at spalte bændlet ledning for ledning ca. 4-5 cm og afisoler de yderste 5-6 mm af alle ledninger.
Sno korerne i de enkelte ledere sammen så ingen kore stritter.
Fortin disse ender.
Hold stikket fast i den viste stilling og
Monter de enkelte ledninger ved lodning som tegningen viser.
Fastgør DIL-stikket i lakridsbændlets anden ende. Obs HAK
Saml hætten over stikket.

DIL-stikket samles lettest i en skruestik, hvor du har lagt en træliste mellem de to rækker ben. Der klemmes altså mellem træliste og stikkets flade låg. Bændlet føres ud gennem hættens sidehul og forbindelseskablet er klar til afprøvning.
ADVARSEL: Sluk for datamaten. Der må ikke forbindes noget til datamaten mens der er spænding på apparatet. Når stikket er sat i datamaten og kablet forbundet til prøvepladen, kan der tændes. Hvis forbindelsen er i orden, kan dioderne stå med svagt lys.
Indtast nu:

```
Poke $dd03,$ff <RETURN>
Poke $dd01,$ff <RETURN>
```

- og dioderne skal lyse tydeligt.

```
Poke $dd01,$00 <RETURN>
```

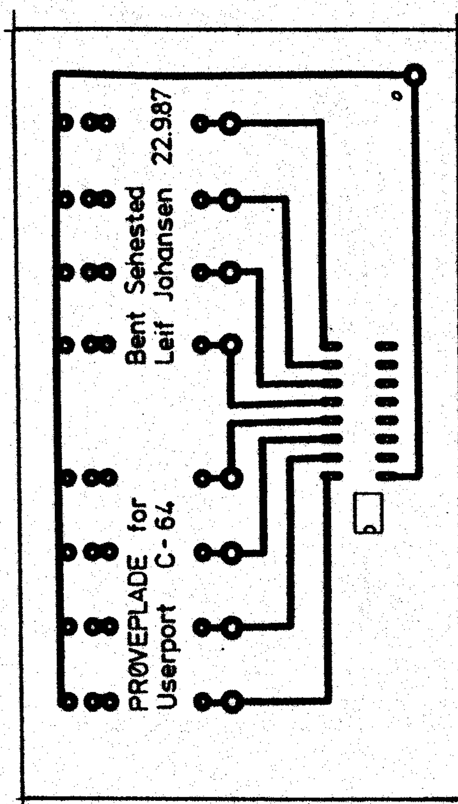
vil slukke alle LED. Kan du efter disse få oplysninger få LED til at lyse i forskellige kombinationer ?

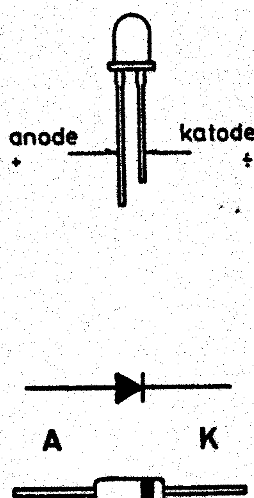
PRINTFREMSTILLING.

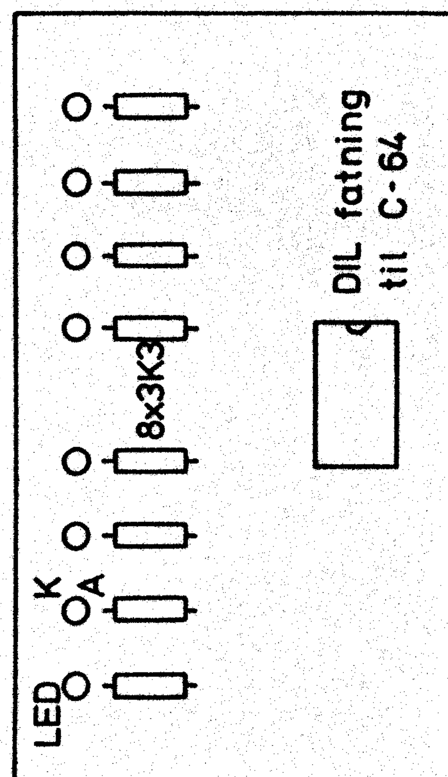
Det valgte kredsløb foreligger som en positiv på grafisk film. Glasfiberpladen kan forberedes ved rensning og påsprøjtning af den lysfølsomme fotoresist Positiv 20. Denne behandling må ikke foregå i sollys. Lad pladen tørre i støvfrit rum ca. 8 timer ved rumtemperatur; den skal ligge vandret under tørringen. Herefter er pladen klar til belysning.
En anden mulighed er at købe færdigpræpareret fotoprint (enkeltsidet). Det leveres med mørkelægningspapir påklistret den følsomme plade.
Til belysning skal bruges en UV-lampe. En god løsning er en 125 watt kviksølvlampe fra Philips (HPR 125 W) ; hertil skal bruges en drosselspole BHL 125L12 og en porcellænsfatning E 27. Lampen har en opvarmningstid på ca 10 min. Den skal hænge 50 cm over glasfiberpladen med den positive film. Brug en glasplade for at holde kontakt mellem film og glasfiber. Med denne opstilling er belysningstiden 6 min. Herefter skal hinden fremkaldes. Du kan bruge SENO develop 112 og endelig afskylning med vand. Selve ætsningen kan foregå med brintoverilte og opløsningen fremstilles sådan:

200 ml konc. saltsyre (39 %) hældes langsomt i
800 ml vand fra vandhanen.
30 ml brintoverilte (35 %) tilsættes.

Nu er blandingen aktiv i ca 1 time - herefter kan den reaktiveres med 30 ml brintoverilte. Printpladen skydes forsigtigt ned i opløsningen og den bevæges jævnt med en træpind eller med en plasticpincet. - Pas på sprøjt - Når pladen er ætset, tages den op og skylles med vand. Ætsebadet må ikke hældes bort efter brug. Det kan opsamles på plasticbeholder og afleveres til f.eks. den lokale farvehandler eller den kommunale depotplads. Husk at oplyse modtageren om dunkens indhold.



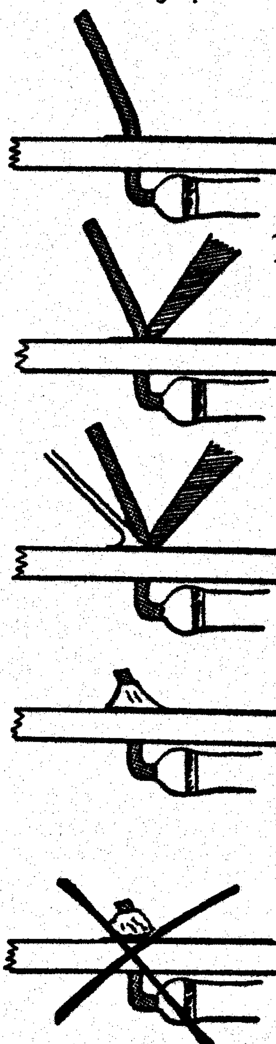




LODNING

For at opnå et godt resultat med et elektronisk byggesæt er det vigtigt, at ALLE lodninger udføres korrekt. En korrekt lodning opnåes med den rigtige fremgangsmåde med lidt træning.

FREMGANGSMÅDE.



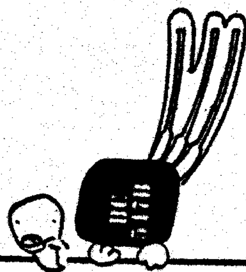
1) Sæt komponenten i printet og buk benene lidt om, så komponenten sidder fast.

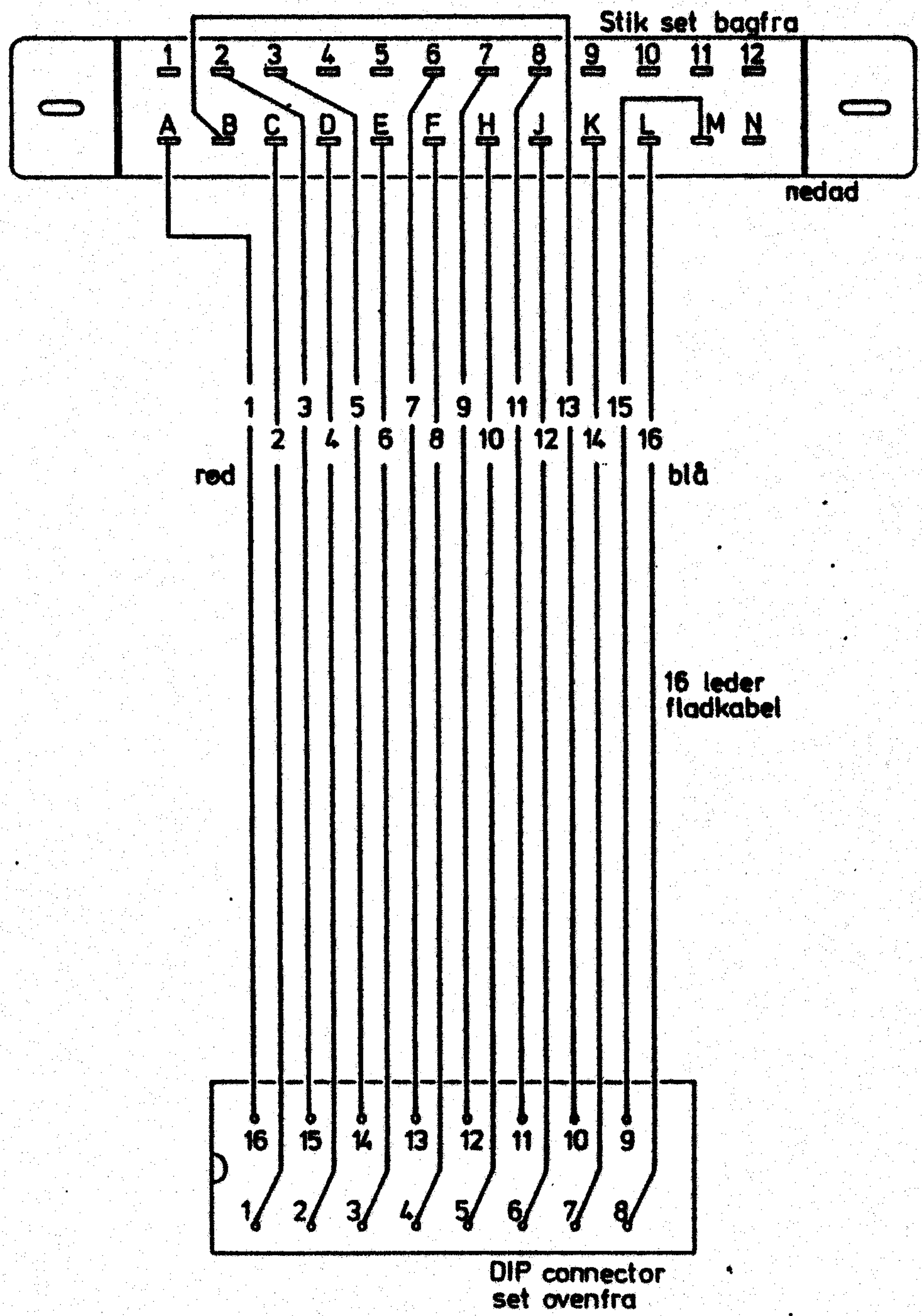
2) Sæt den varme loddespids på loddestedet, således at både loddeø og komponentben opvarmes samtidigt.

3) Tilfør derefter loddetin. Når lodningen er afkølet, klippes komponentbenene af 2 til 3 mm fra printet.

4) En god lodning med den rigtige mængde tilført loddetin vil være konkav med en glat overflade. Tilføres for meget loddetin, fås en kuglerund lodning.

5) Varmes komponentben og loddeø ikke op samtidig, fås en "kold-lodning". Loddetinnet danner da ikke ordentlig kontakt mellem komponentben og loddeø. En koldlodning rettes ved at varme loddestedet op igen, og evt. tilføre lidt friskt loddetin.

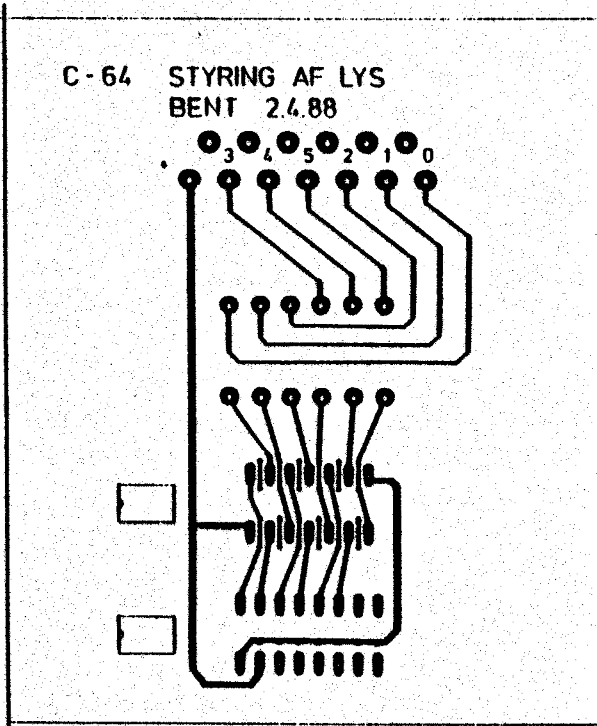


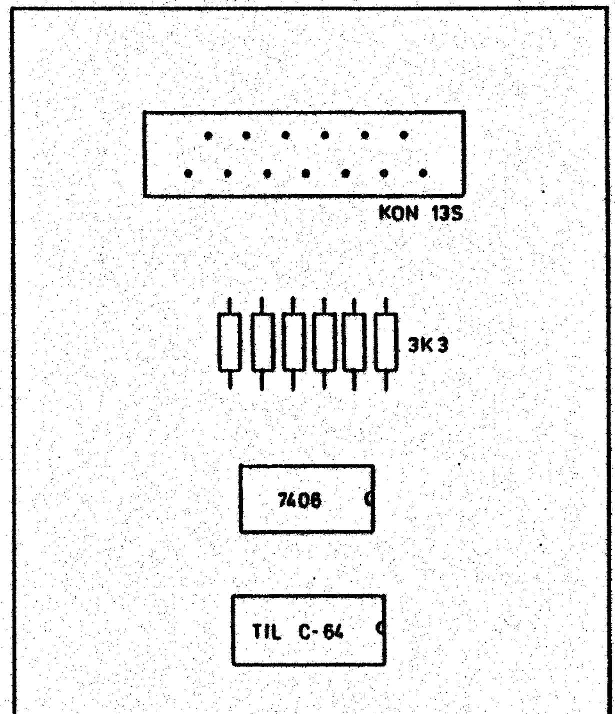

Stik set bagfra
1 2 3 4 5 6 7 8 9 10 11 12
A B C D E F H J K L M N
nedad
1 3 5 7 9 11 13 15
2 4 6 8 10 12 14 16
rød
blå
16 leder
fladkabel
16 15 14 13 12 11 10 9
1 2 3 4 5 6 7 8
DIP connector
set ovenfra

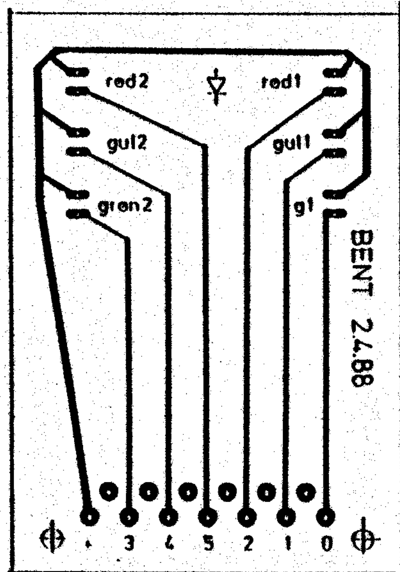
Fremstilling af det elektroniske lyskryds.

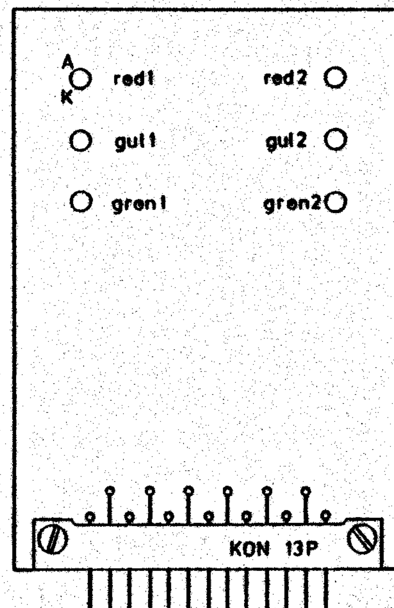
Herunder ses printudlæg og monteringsplan for bundplade og opsats. Forbindelsen sker via et han/hunstik KON 13P/13S. Forbindelseskablet til C-64 er det samme som til lysdioderne. På bundpladen ses en driver, der skal beskytte datamaten under drift, idet du kan sætte flere lysdioder på opsatsen. Du må dog selv ændre opsatsen.

Stykliste:
1 IC 7406 Hexinverter
1 stik Kon 13P/13S
2 LED rød, gul, grøn
1 sokkel DIL 14
1 sokkel DIL 16
6 modstande 3K3



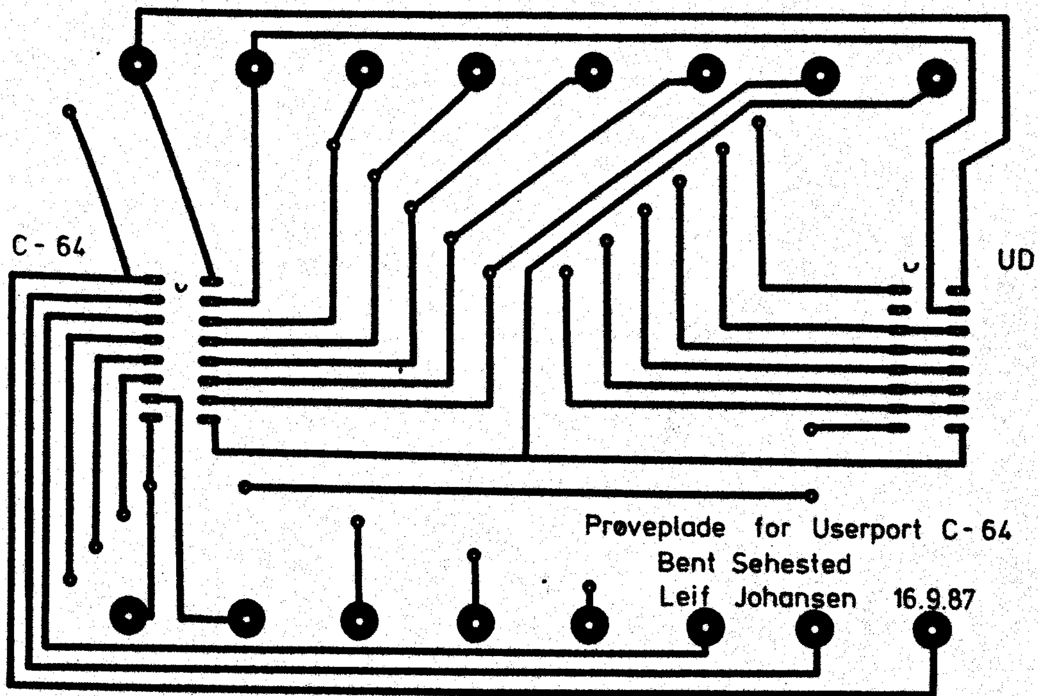


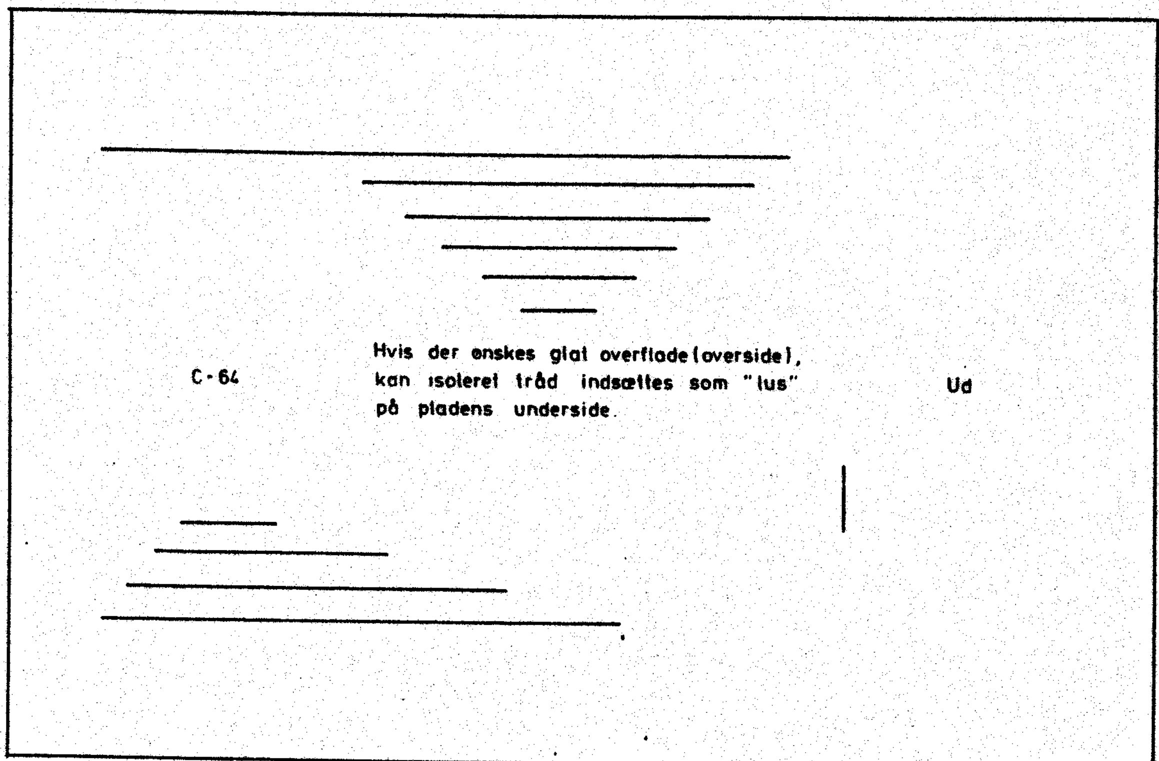




Hvis du vil i kontakt med andre terminaler end portb, kan du bygge denne plade og forbinde den med datamaskinen via båndkablet.
Tegningen på dette blad viser over- og underside. På oversiden er aftegnet de "lus", der skal fastloddes. Hvis du ønsker en glat overside, kan du lave lus af isoleret monteringstråd og fastlodde dem på pladens underside.
På næste blad ses arket med den tekst, der bør findes på den færdige plades overside. Den letter korrket forbindelse.

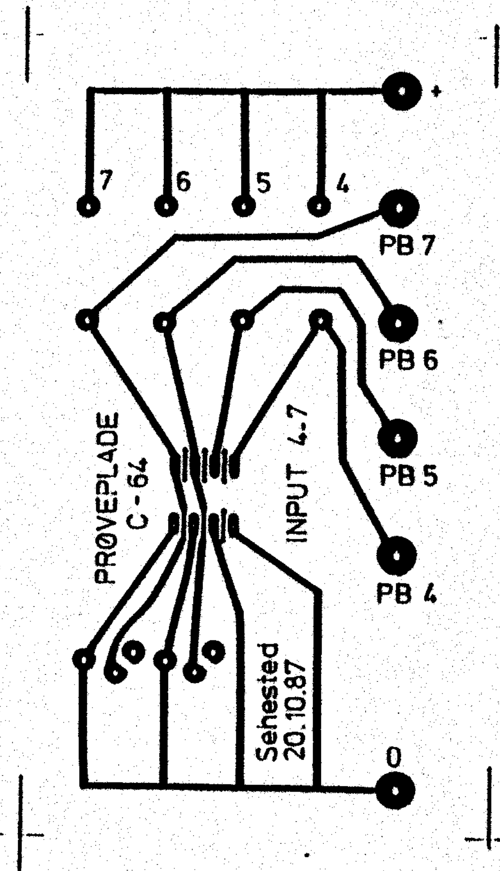
Der bruges	2 sokler med 16 ben DIL	elfi 294.3016.0
	16 printstifter	elfi 450.2010.0
	minikrokodillenæb hertil	elfi 410.0350.0
eller	16 rørbøsninger	elfi egenproduktion
	bananstik 2,6 mm hertil	elfi 410.0010.0

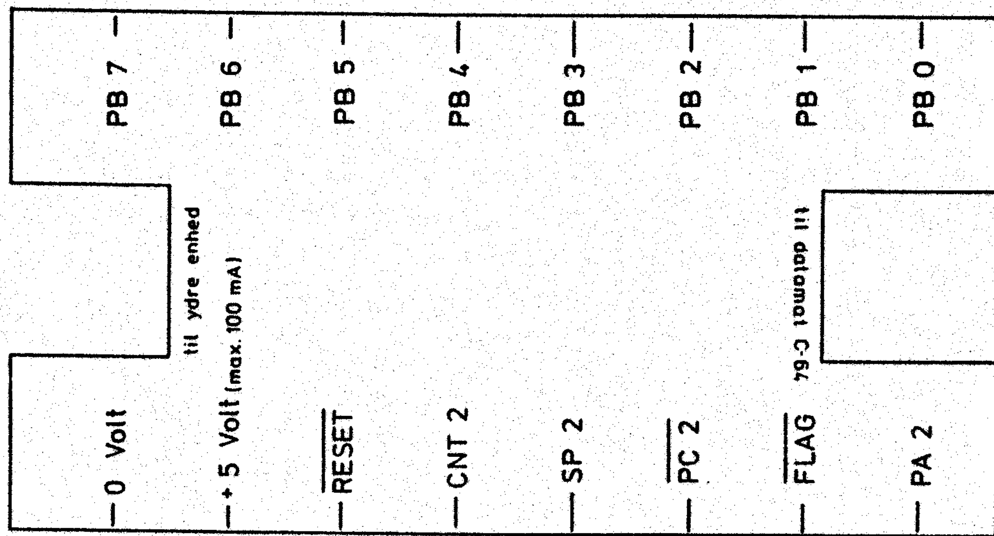




Her er printudlæg for en prøveplade, der kan vise brug af kontakter og trykknapper.

2 stk DIGITAST 1x1 cm for printmontage
1 stk 4-polet dipswitch elfi 425.0004.0
4 stk kulmodstand 3k3
rørbøsninger eller printstifter

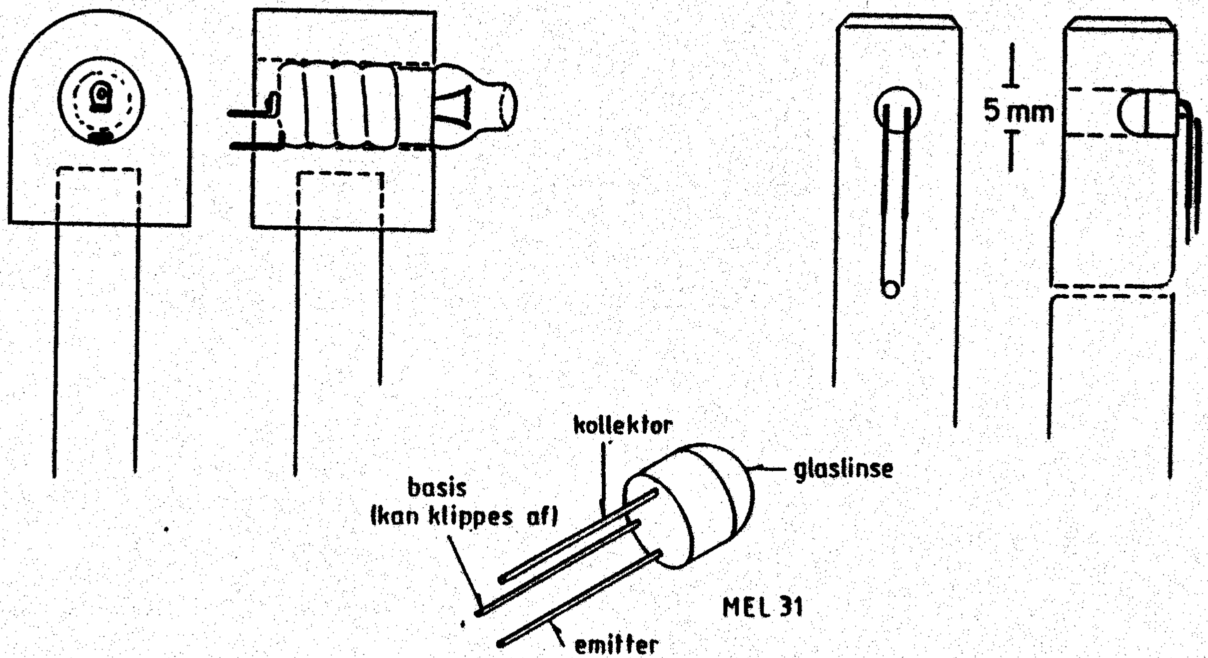




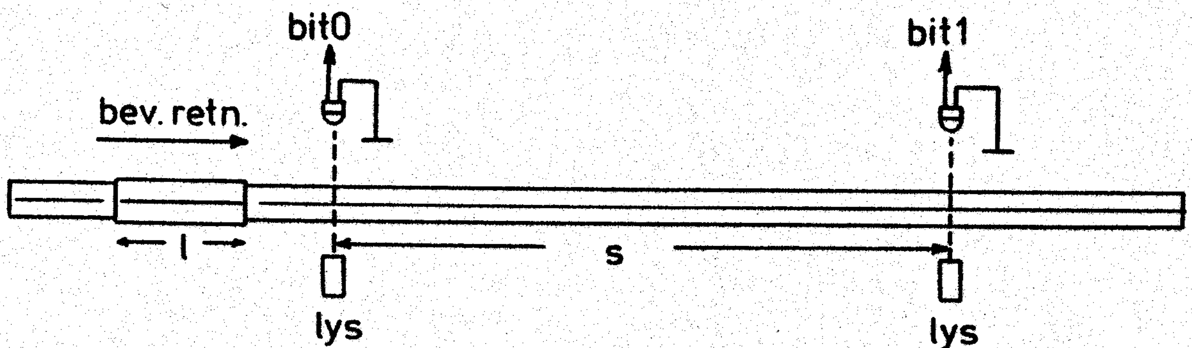
Skal du fremstille en lysbro, kan det gøres v.hj.af 4 stk rundstok og 2 stk træklods.
Fototransistoren MEL 31 sættes i et hul i stokken og dens ben overtrækkes med krympeflex; - brug rød monteringstråd til collektorben og sort tråd til emitterben; - det letter oversigten, hvis du gennemføre dette farvevalg. Altså: sort ledning til stel, rød ledning til + (kaldes ofte "den varme ende").
I opstilling til accelerationsmålinger forbindes 1. fototransistor til bit0, 2. fototransistor til bit1.

NB ! Strømforsyningen til dværglampe med linse må ikke tages fra datamaten. Du må bruge elementer, f.eks. batterikassen fra Ficher-technik.

Materialer: Fatning E-10 messing elfi 430.8120.0
Fototransistor MEL 31 elfi 224.0031.0
Dværglampe med linse, 2,3 v E-10



måling på luftpudebænk



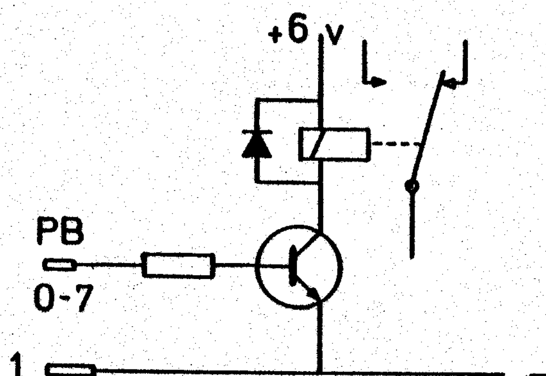
Indtil nu har du tændt og slukket lysdioder med datamaten. I princippet er der ingen forskel på den opgave og at styre en el-motor - dog er der 2 væsentlige begrænsninger:

1) datamaten kan ikke levere den strøm, der skal trække et relæ.
2) relæet må ikke sluttes til userportens strømforsyning.

I portb kan du ændre spændingen mellem 0 volt og ca. 5 volt ved at stille et bit lav eller høj. Den strøm, der maximalt må hentes fra datamaten er 100 mA.

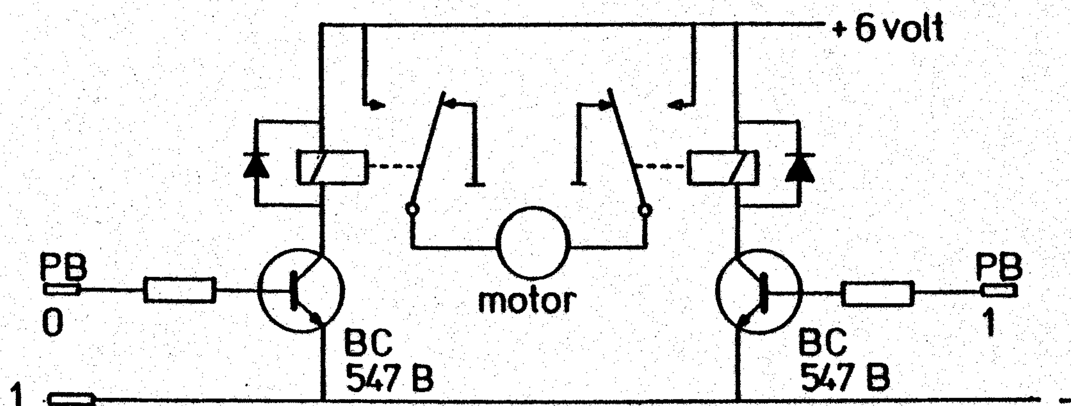
Første forhindring klares med et transistorforstærkertrin.
Anden forhindring omgås med særskilt strømforsyning.

Hvis vi ønsker at starte og stoppe en motor, kan vi klare os med 1 bit pr. motor. Vil vi også kunne styre motoren højre/venstre om, må vi bruge 2 bit pr. motor. - Princippet ses af diagrammet her:



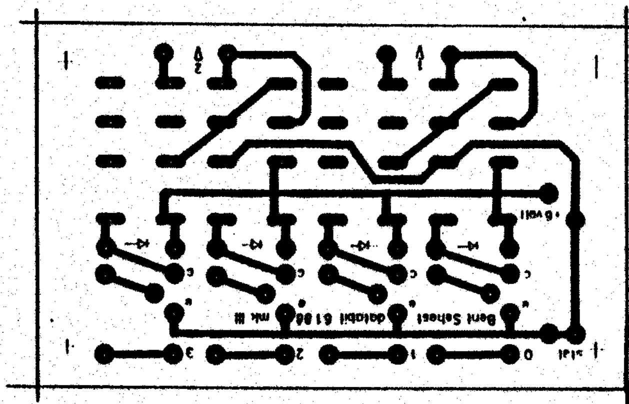
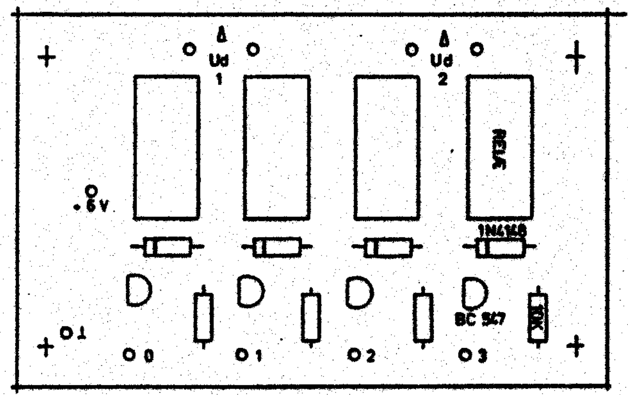
Tegningen viser et forstærkertrin med en transistor, hvor basis er forbundet til portb, f.eks. bit0. Transistoren sidder som kontakt for en relæspole, der er sikret med en diode. Når bit0 sættes høj, går basis mod + og transistoren begynder at lede. Strømmen gennem spole og transistor får relæet til at trække kontakten til sig. Du kan høre et lille klik fra relæet, når det sker. Kontakten bliver stående i denne stilling, indtil bit0 sættes lav igen.

Hvis du vil bruge konstruktionen til at styre jævnstrømsmotorens omløbsretning, må du lade 2 porte arbejde sammen, som det næste diagram viser. Bemærk: + 6 volt er fra en ydre strømforsyning.

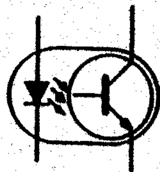


Denne løsning anvendes i printet "Databil mk III", der er indbygget i dozer og i lastbil.
Se printudlæg på næste side. Her bruges bit0 og bit2 til stop/start; bit1 og bit3 til højre/venstre for de to motorer.

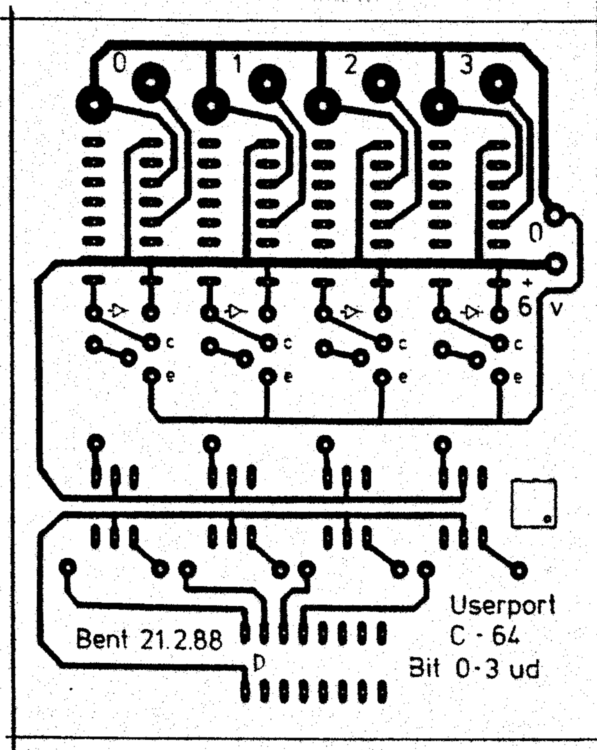
Printudlæg og monteringsplan for lastbil:

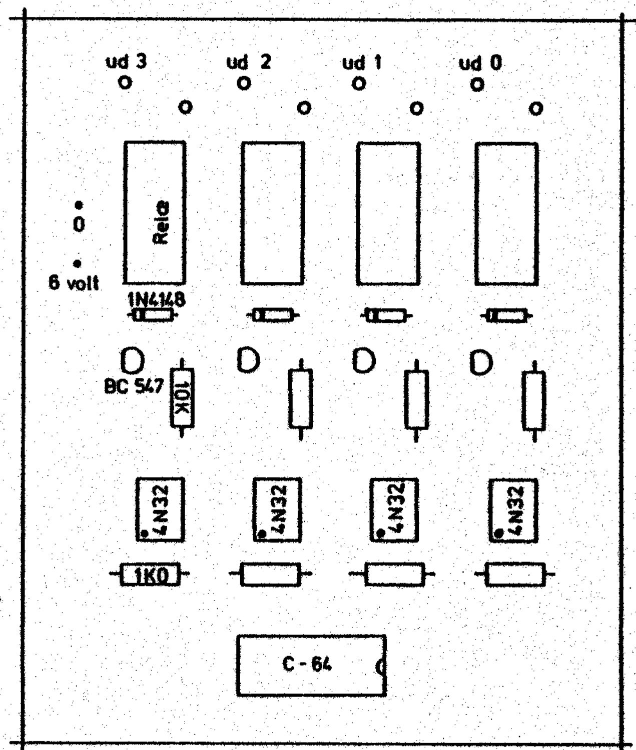



For helt at undgå forbindelse mellem datamat og ydre kredse, har vi i den næste konstruktion valgt at indsætte en optokobler i hver ledning fra userporten. En optokobler er en sammenbygget enhed, hvor en lysdiode er anbragt lige foran basis på en fototransistor.

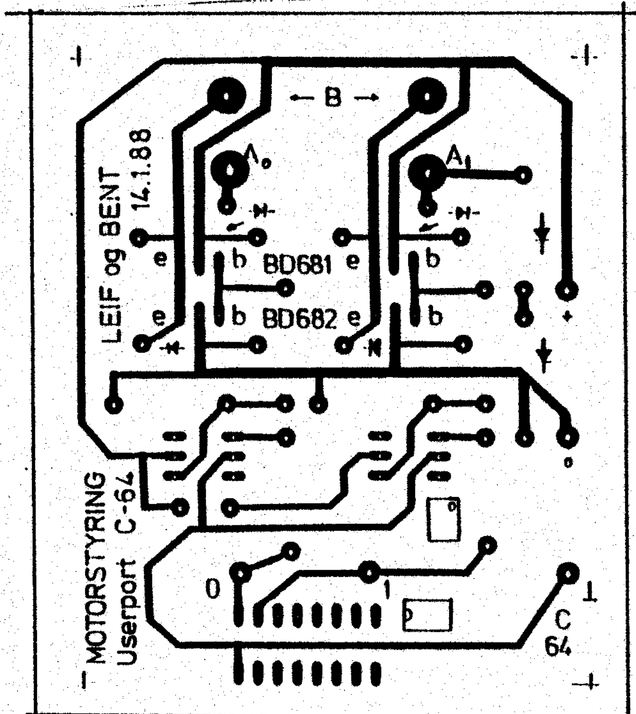


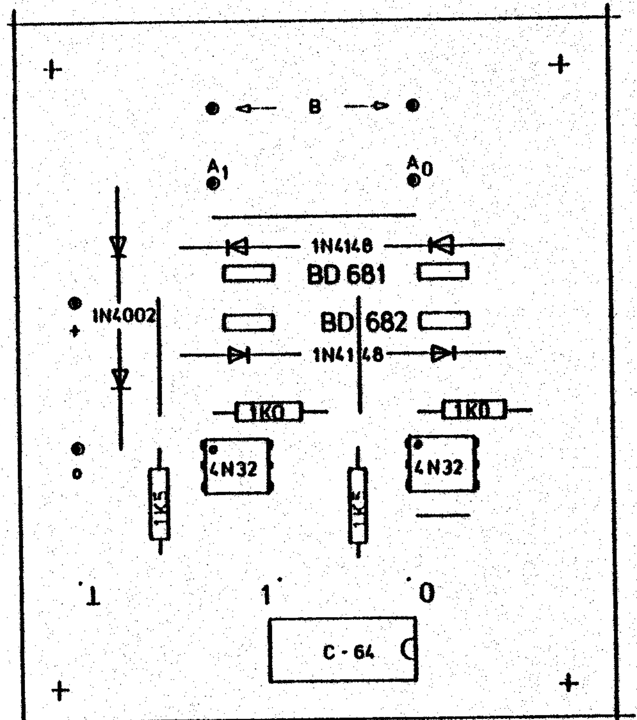
Når lysdioden er tændt, vil lyset åbne fototransistoren, der så leder. Slukkes dioden, lukker transistoren øjeblikkelig. Ved denne løsning er der ingen elektrisk forbindelse mellem datamat og ydre strømforsyning. Det er lyset fra LED, der overfører informationerne. Optokobleren her er udført som lille enhed med 6 ben DIL; så den passer direkte i en sokkel. Ved denne løsning benyttes 4 bit og 4 relæer. Pladen kan altså bruges til at styre 2 motorer højre/venstre og stop/start.
BEMÆRK ! Relæer må ikke bruges i forbindelse med styringsopgaver, skrevet i maskinkode. Kontakterne i relæer kan ikke følge med. Her må anvendes styring v.hj.a. transistorer i stedet.



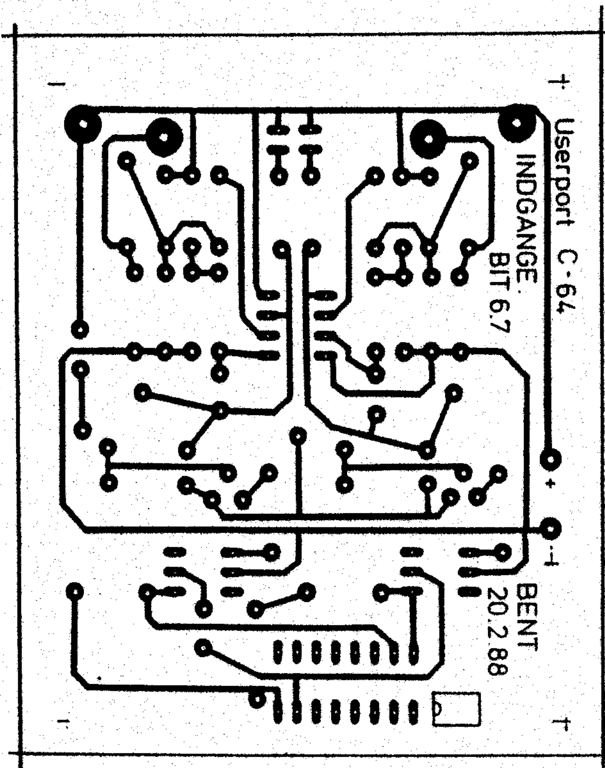


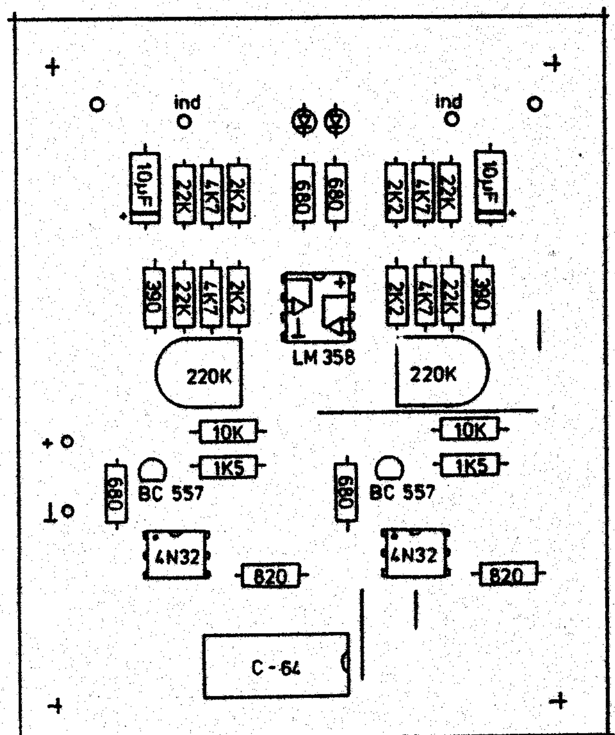
I løsningsforslaget herunder er anvendt darlingtontransistorer, men almindelige effektransistorer BD135/136 kan også bruges.



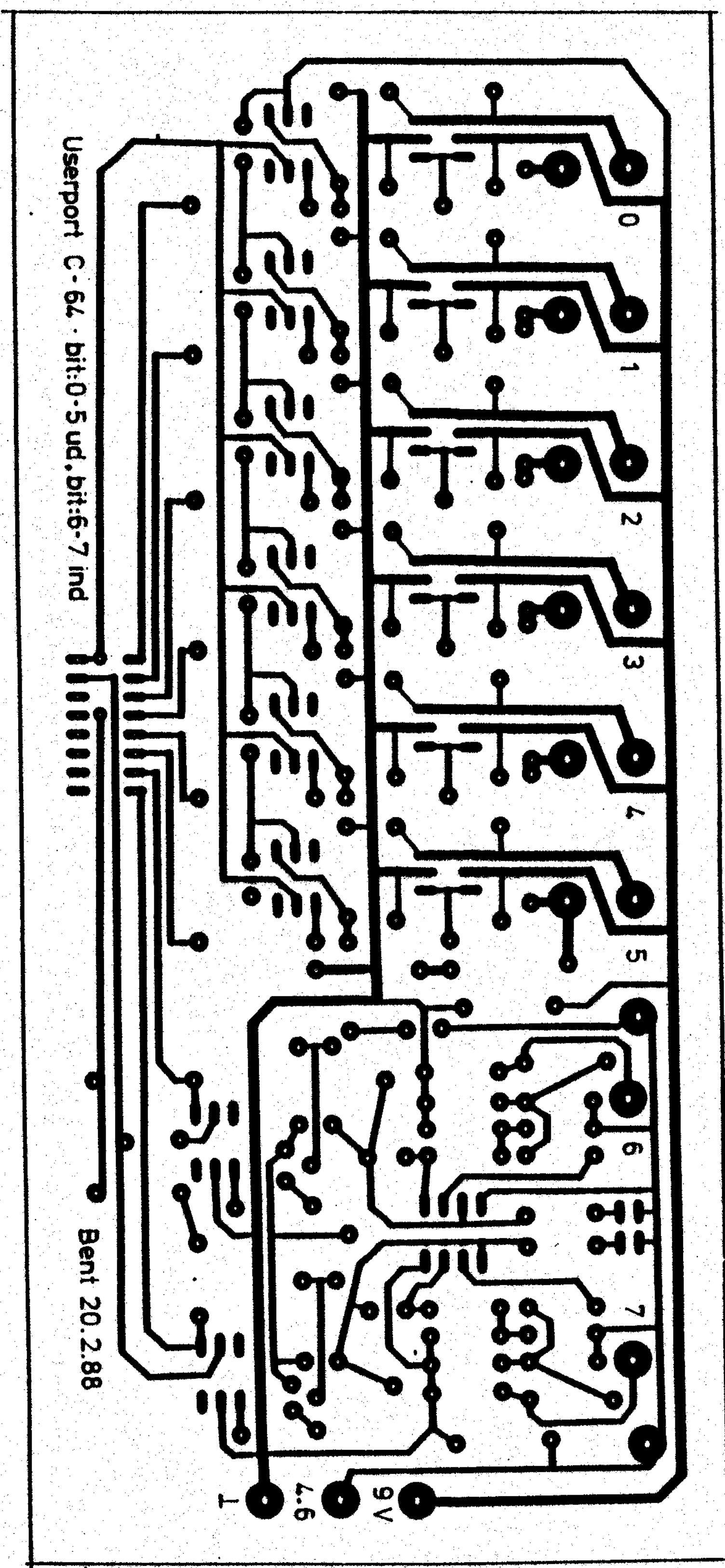


Skal oplysning om kontakters stilling opsamles af datamaten, kan du få brug for 2 indgange, der via en fælles IC med to operationsforstærkere kan registrere en overgang mellem høj og lav.





Her findes en kombineret printplade med 6 udgange og to indgange. Komponenterne er de samme som ved de separate opstillinger, så styklisterne kan fremstilles herudfra.

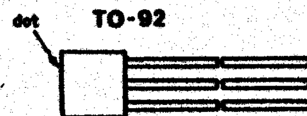


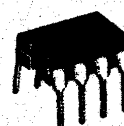
Materialeliste:

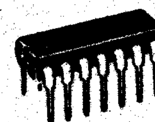
Relæprint (pr.relæ)
Relæ NEC MR62 428.3062.0
Diode 1N4148 202.4148.0
Trans BC547 261.0547.2
Modst 10K
Printstifter eller
Rørbøsninger

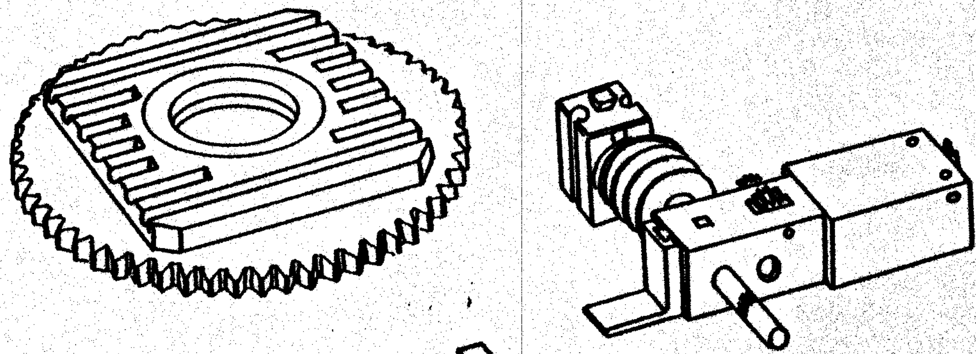
Transistorprint
2 Diode 1N4002 202.4002.0
4 Diode 1N4148 202.4148.0
2 Optok 4N32 232.0432.0
2 Trans BD681 262.0681.0
2 Trans BD682 262.0682.0
2 Modst 1K0
2 Modst 1K5
2 Sokkel DIL6 294.3006.0
1 Sokkel DIL16 294.3016.0
Printstifter eller
Rørbøsninger

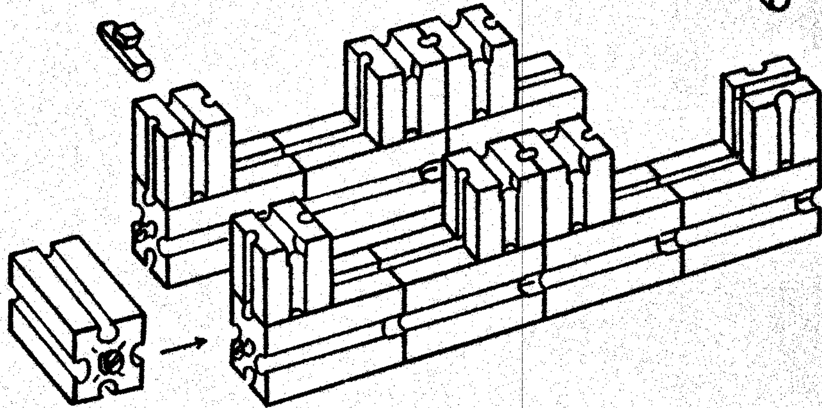
Print for indgang
1 Opamp LM358 300.0358.0
2 Optok 4N32 232.0432.0
2 Trans BC557 261.0557.2
2 LED, rød 3mm 221.3002.0
2 Elko 10uF/25 170.2106.0
2 Modst 390
4 Modst 680
2 Modst 820
2 Modst 1K5
4 Modst 2K2
4 Modst 4K7
2 Modst 10K
4 Modst 22K
2 Potm 220K 121.3224.0
1 Sokkel DIL8 294.3008.0
1 Sokkel DIL16 294.3016.0
Printstifter eller
Rørbøsninger

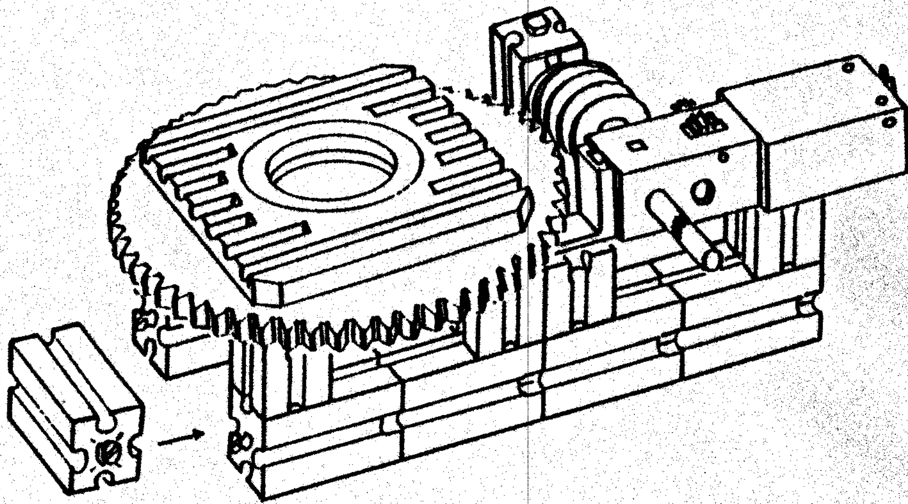


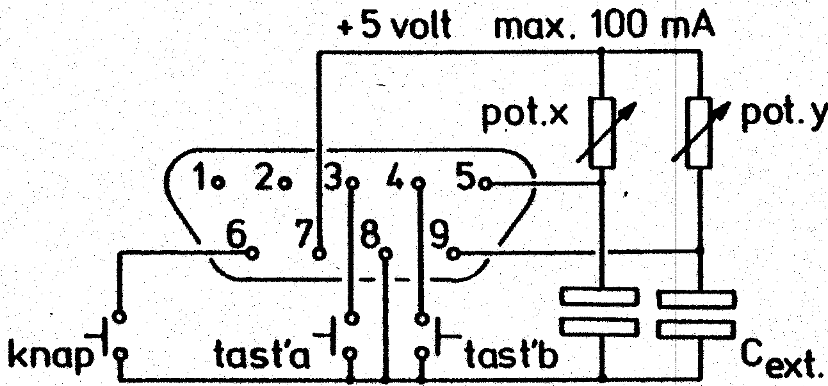




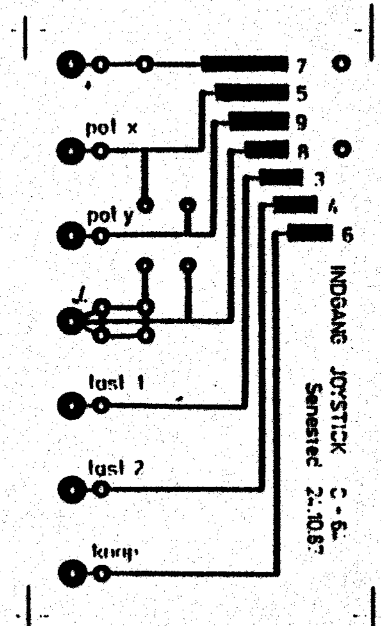


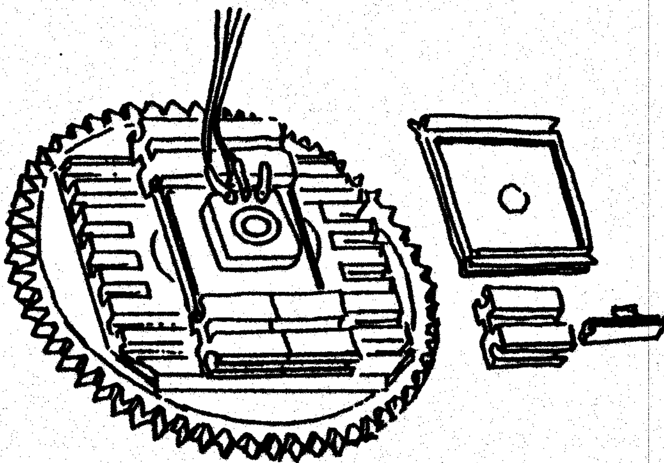






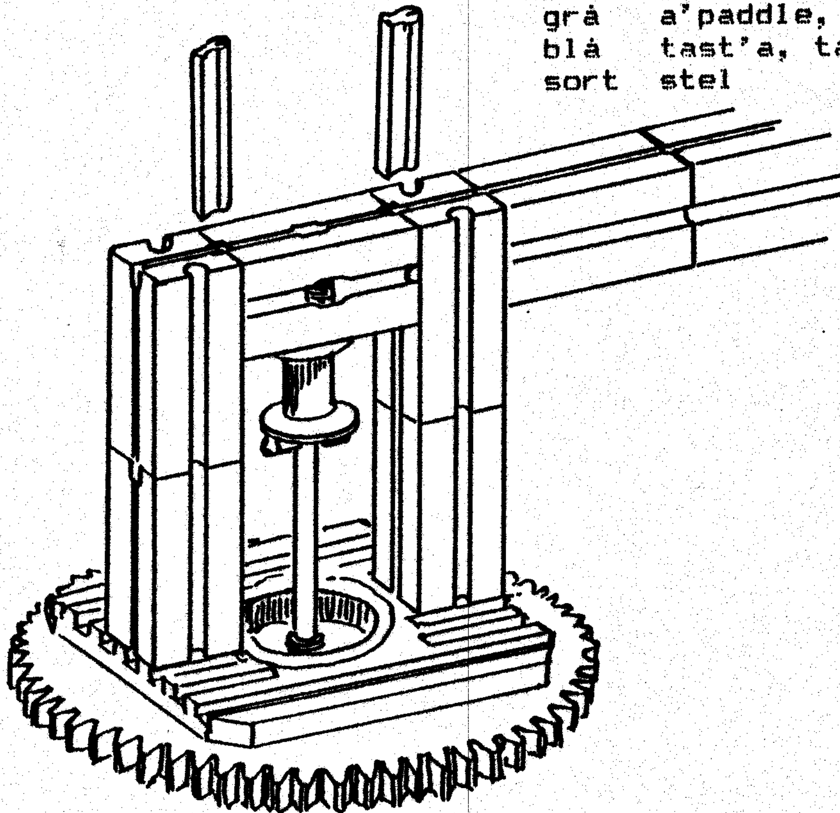
Joystickport set udefra

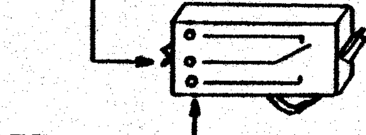


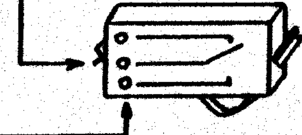


Når du samler stikket,
vil det være klogt, at
vælge farver på kablerne,
så forbindelserne kan
kendes.
De to kondensatorer,
C ext. er begge 100 nF
De kan monteres inde i
stikket.
Farvekoden der anvendes her:

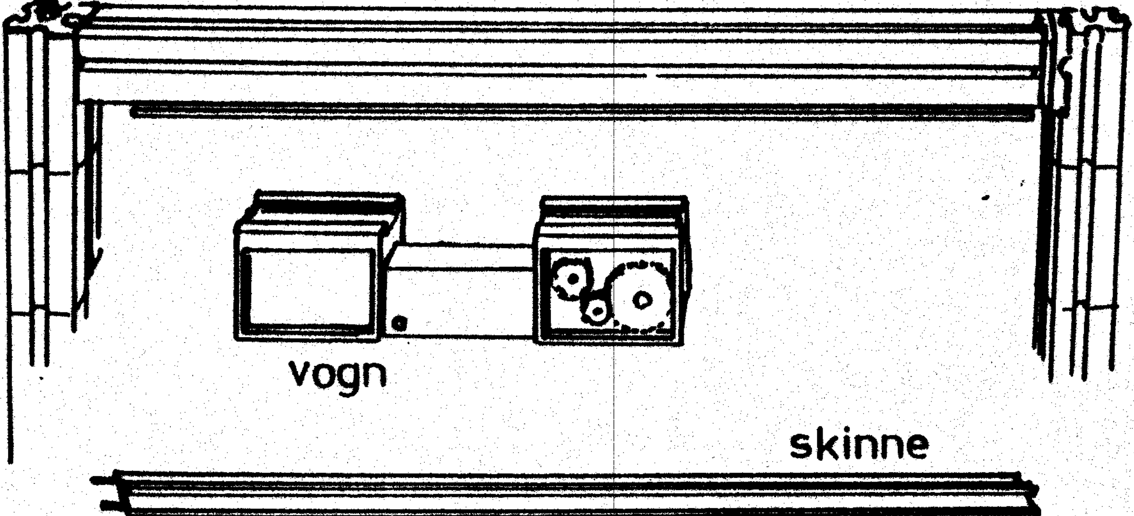
rød	+ 5 volt (hunstik)
grå	a'paddle, b'paddle
blå	tast'a, tast'b
sort	stel

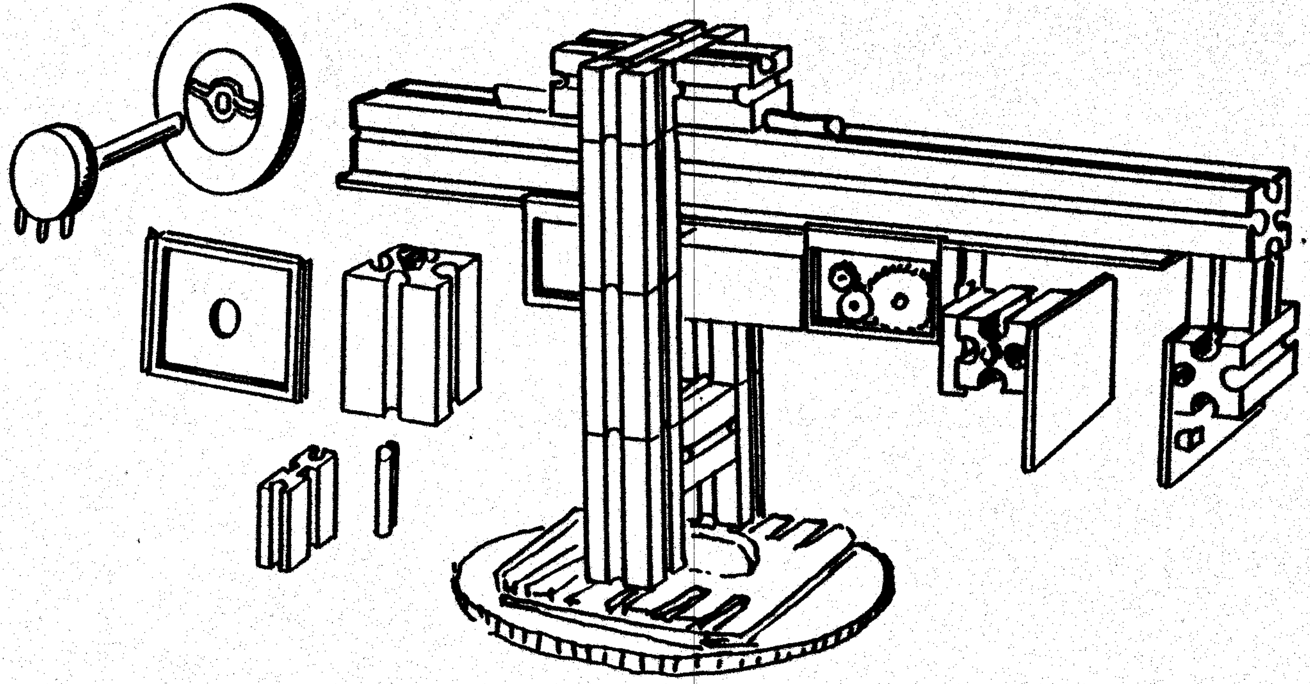




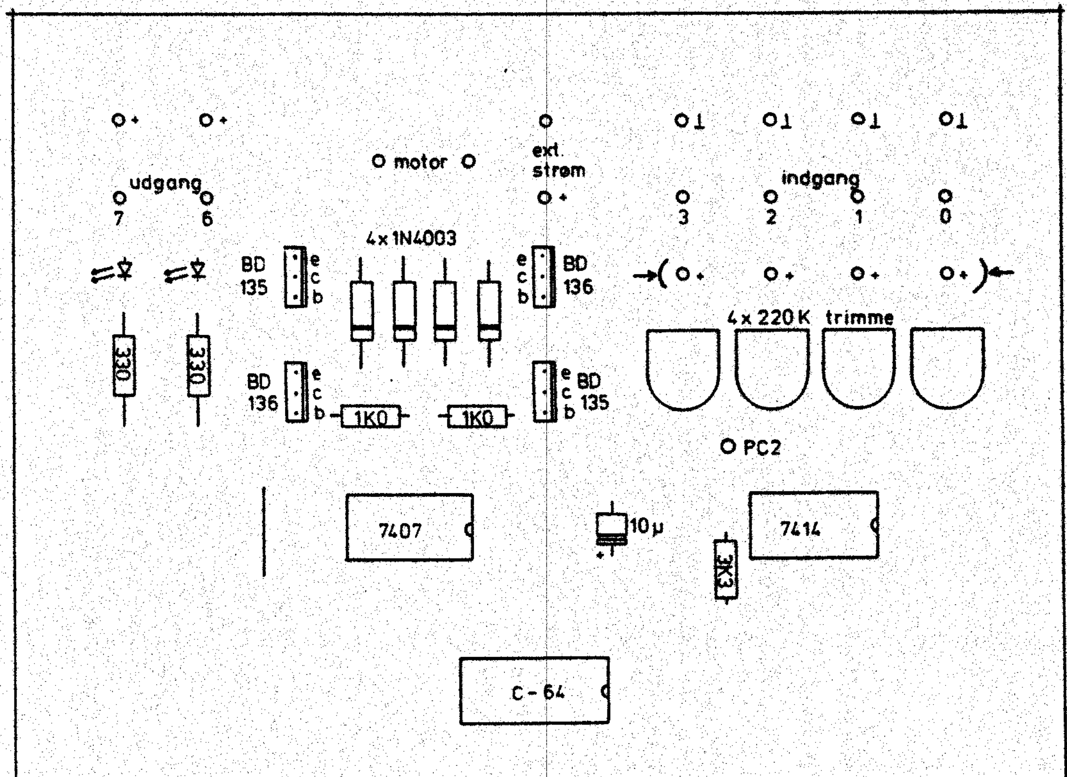


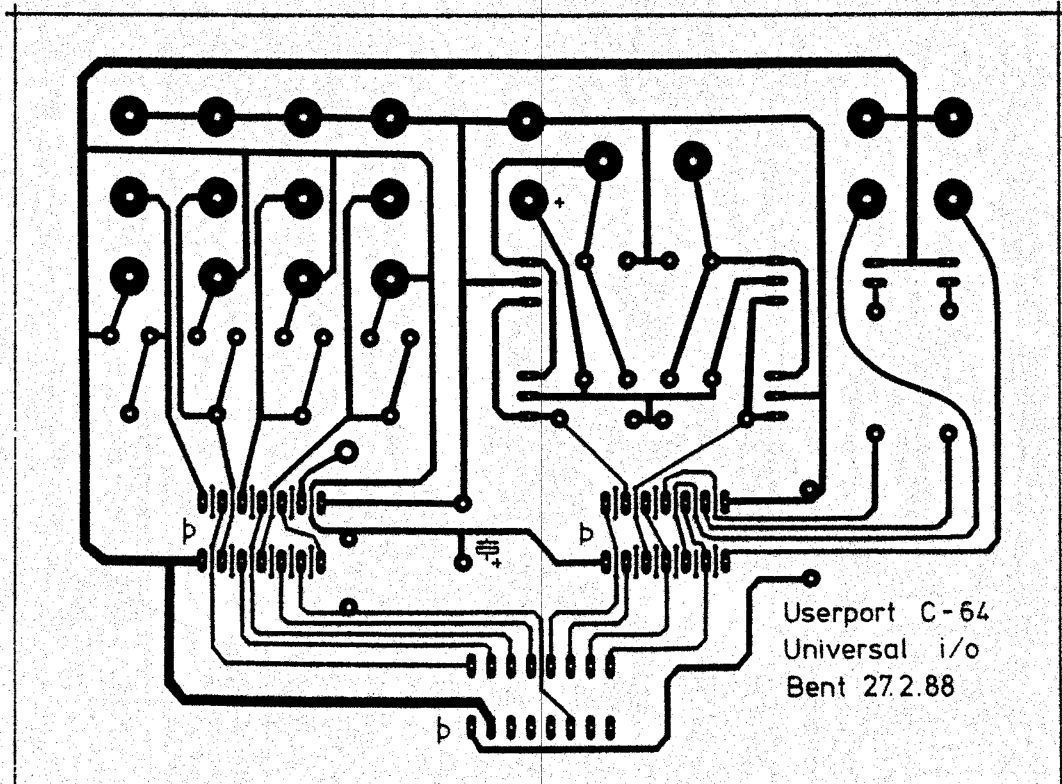
stel





Universalprint bit0-bit3 ind
bit4 bit7 ud



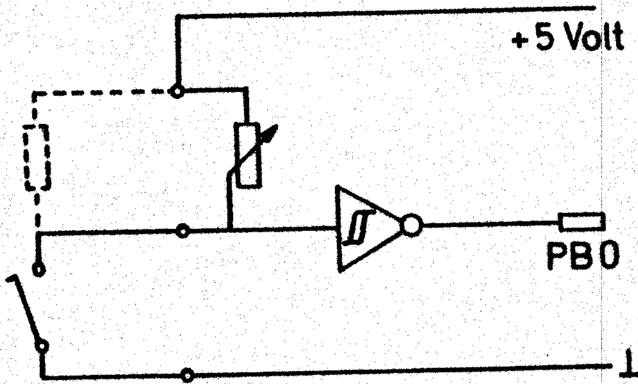


stykliste næste side

Universalprint - beskrivelse og stykliste.

Konstruktionen rummer 4 indgange, der er lagt til de 4 mindste bit og 2 udgange til bit4 og bit5. Herfra skal motorstyringen ske. De 2 højeste bit anvendes ikke her.

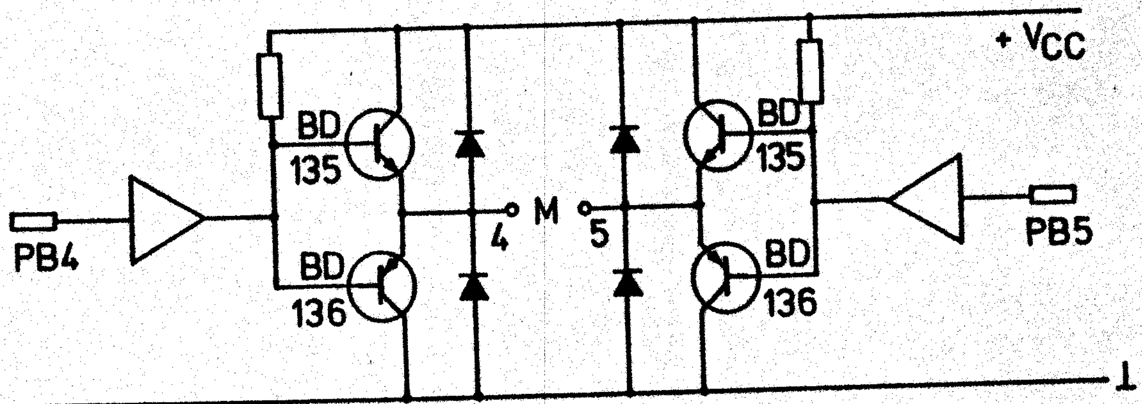
Indgange.



Alle indgange er forsynet med et potentiometer, så følsomheden kan reguleres. Der er på tegningen antydet en ydre modstand, hvis den ønskes i stedet for potmeteret. For at gøre spændingsvariationen veldefineret, er signalet ført til en Schmitt trigger inden den lander på portens bit0 - bit3.
Når kontakten er åben, er PB0 høj. Lukkes kontakten, ligger spændingsfaldet over modstanden, og PB0 går lav.
Datamaten kan dermed registrere, om kontakten er åben eller lukket.

Udgange.

PB4 og PB5 er ført til hver sin driver. Herfra ledes signalet til basis af de komplementære effekttransistorer, der åbner og lukker eftersom signalet er højt eller lavt.

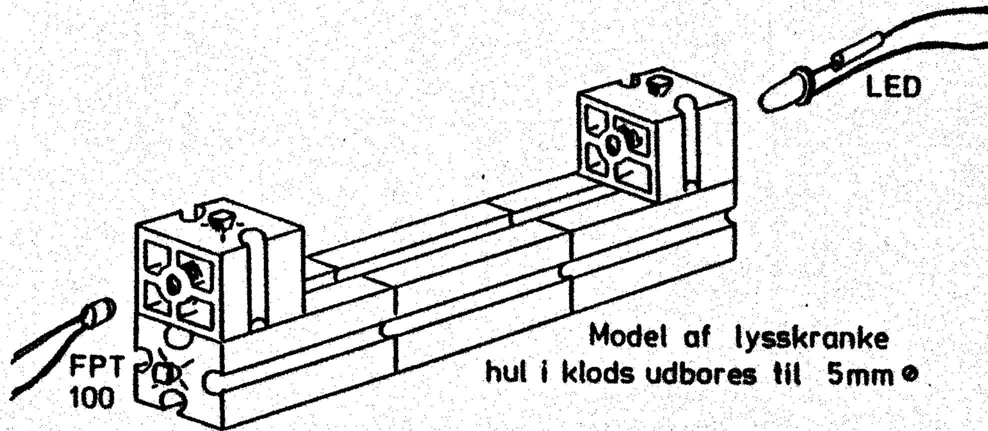


Går PB4 høj, åbner BD 135 og M4 vil gå mod Vcc. Er PB5 lav, åbner BD 136 og M5 går mod 0. Spændingsforskellen mellem M4 og M5 er Vcc - 2 diodestrækninger, altså 1,4 volt lavere end Vcc. Denne spænding passer til de små legetøjsmotorer, der laves på en legetøjsfabrik i det midtjyske, men når vi bruger mini-motorer, må vi råde over 5 elementer i en batteripakke for at lave de nødvendige 6 volt.

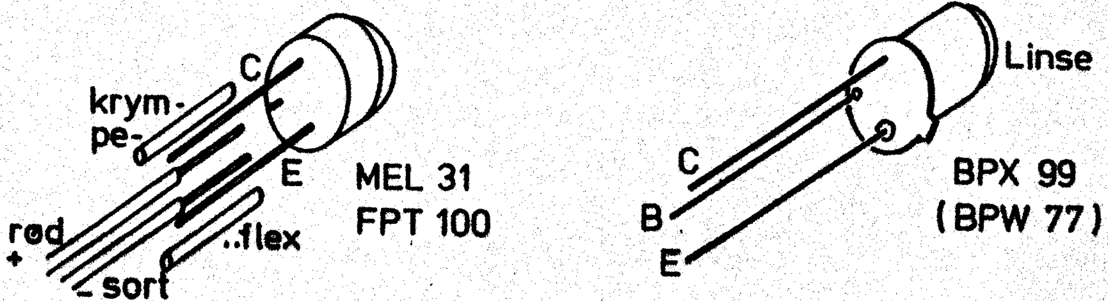
Stykliste for universalprint:

2 BD 135 transist.	elfi	262.0135.0	1 10uF/25 elko	elfi	170.2106.0
2 BD 136 transistor		262.0136.0	2 LED 5mm gul	elfi	221.5004.0
1 74LS07 driver oc		332.0007.0	2 14DIL sokkel		294.3014.0
1 74HCT14 schmitt-tr.		343.0014.0	1 16DIL sokkel		294.3016.0
4 1N4148 dioder		202.4148.0	1 3K3, 2 1K0, 2 K33 modstande		
4 M22 trimpot.meter		121.6224.0	rørbøsninger	elfi	egenprod.

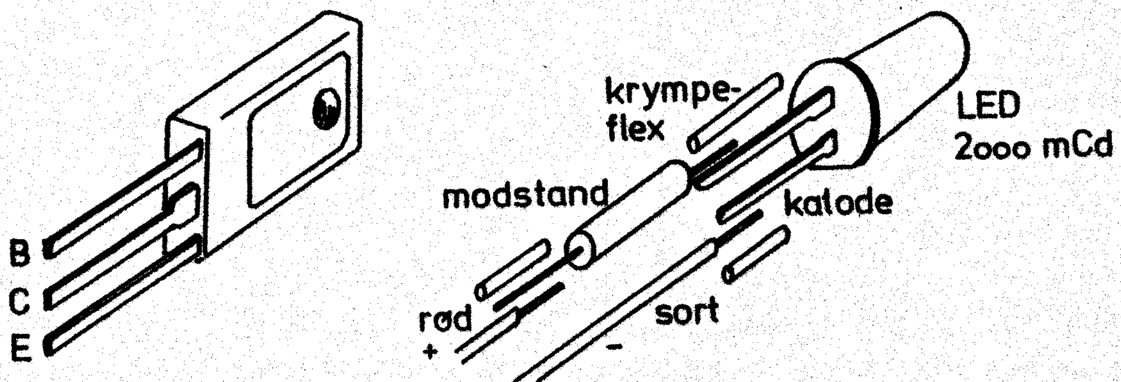
Fremstilling af lysskranke.



Vi anvender en 5mm LED, der kan udsende lys op til 2000 mcd. (elfi 221.5022.0) Lyset koncentreres af plastichuset, hvis top er formet som en samlelinse. Som modtager bruger vi en fototransistor (FPT 100), der kan udstyre indgangen på konstruktionen side $0b nederst. Lysgabet er i forsøgsopstillingen 60 mm og de to klodser skal udbores til 5 mm, - så passer komponenterne stramt i disse huller.



Vi ser først på modtagerdelen: Konstruktionen er vist tidligere med MEL 31, men denne fototransistor er ikke følsom nok til opgaven; - derfor er FPT 100 valgt. Sæt et stykke krympeflex udenom ledningerne og hold fast ved ideen med rød ledning til positiv pol (her collektor). Vil du lave lysskranker, der kræver større følsomhed, kan du anvende fototransistorer eller darlingtonfototransistor med linse i metalhuset.
Styklisten nævner 2, der er følsomme overfor det røde lys LED'er udsender. Benforbindelser kan også ses på skitsen herover.



Nu lyskilden: Som standard for en lysdiode kan du regne med, at den skal genenemløbes af en strøm på omkring 200 mA for at lyse

ordentligt. Spændingsfaldet over dioden er så omkring 2 volt.
Forsøgsopstillingen viste, at lyset var tilstrækkeligt ved 11 mA og med et spændingsfald på 1,8 volt.
Vi råder ikke over 1,8 volt, men 6 volt fra et sæt friske elementer. (ca. 5 volt, når de er brugte) Derfor må der indskydes en modstand, som kan lave et spændingsfald på 4,2 volt ved en strøm på eks. 15 mA. En middelværdi er en modstand på 270 ohm. Lod modstanden på LED's ene ben og træk et stykke krympeflex over det hele. HUSK: rød ledning til anoden. Ved siden af ses benforbindelsen til effekttransistorerne BD 135/136.

Stykliste.

LED udsender 2000 mcd	elfi	221.5022.0
FPT 100 fototransistor	-	224.0100.0
BPW 77A - m linse	-	224.0077.0
BPX 99 darlington -	-	224.0099.0
K27 1/4 w modstand		
3,2 mm Ø krympeflex	-	486.0032.0