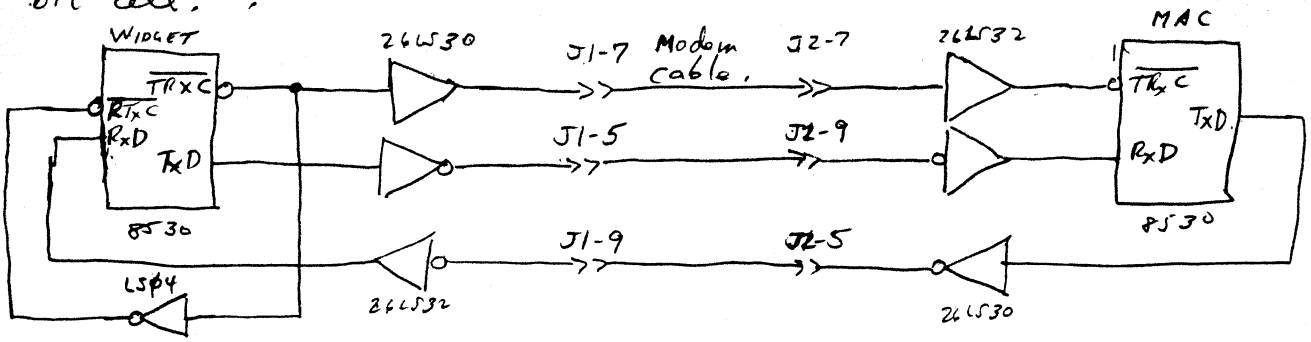
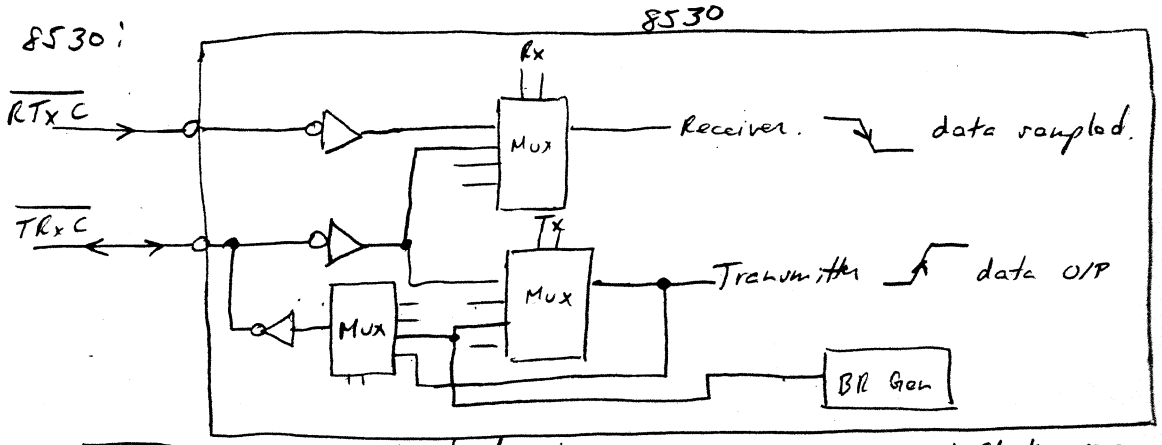
5/29

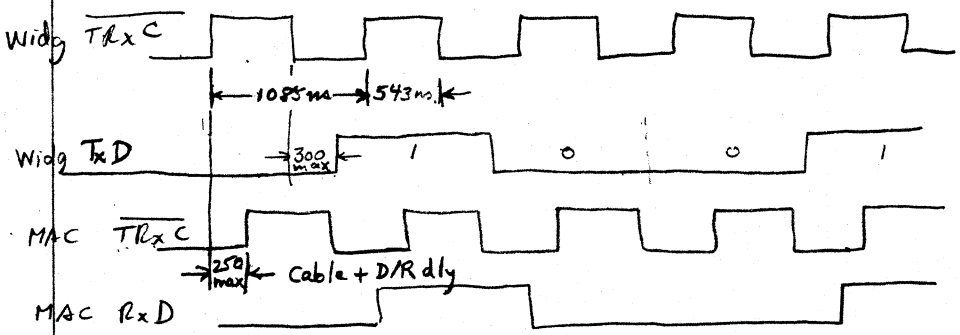
Clocking w/ NRZI or NRZ - gives most margin since sample in middle of bit cell.:



In the 8530:

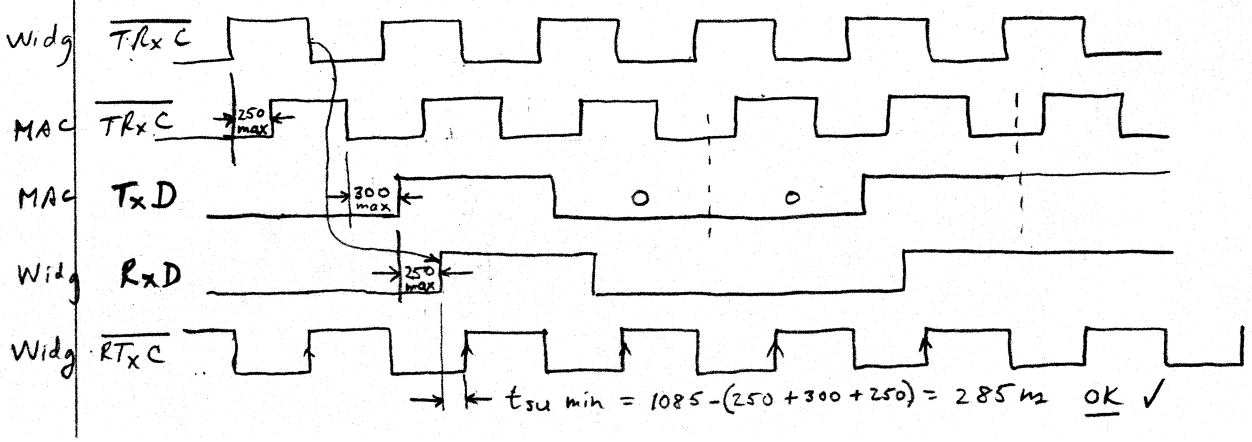


So if use $\overline{TRxC}$ as transmit clock by programming Transmit Clock = BR Gen O/P and $\overline{TRxC}$ = Transmit Clock at Widget end, then data transmission from **Widget → MAC** will work as shown above and in timing below:



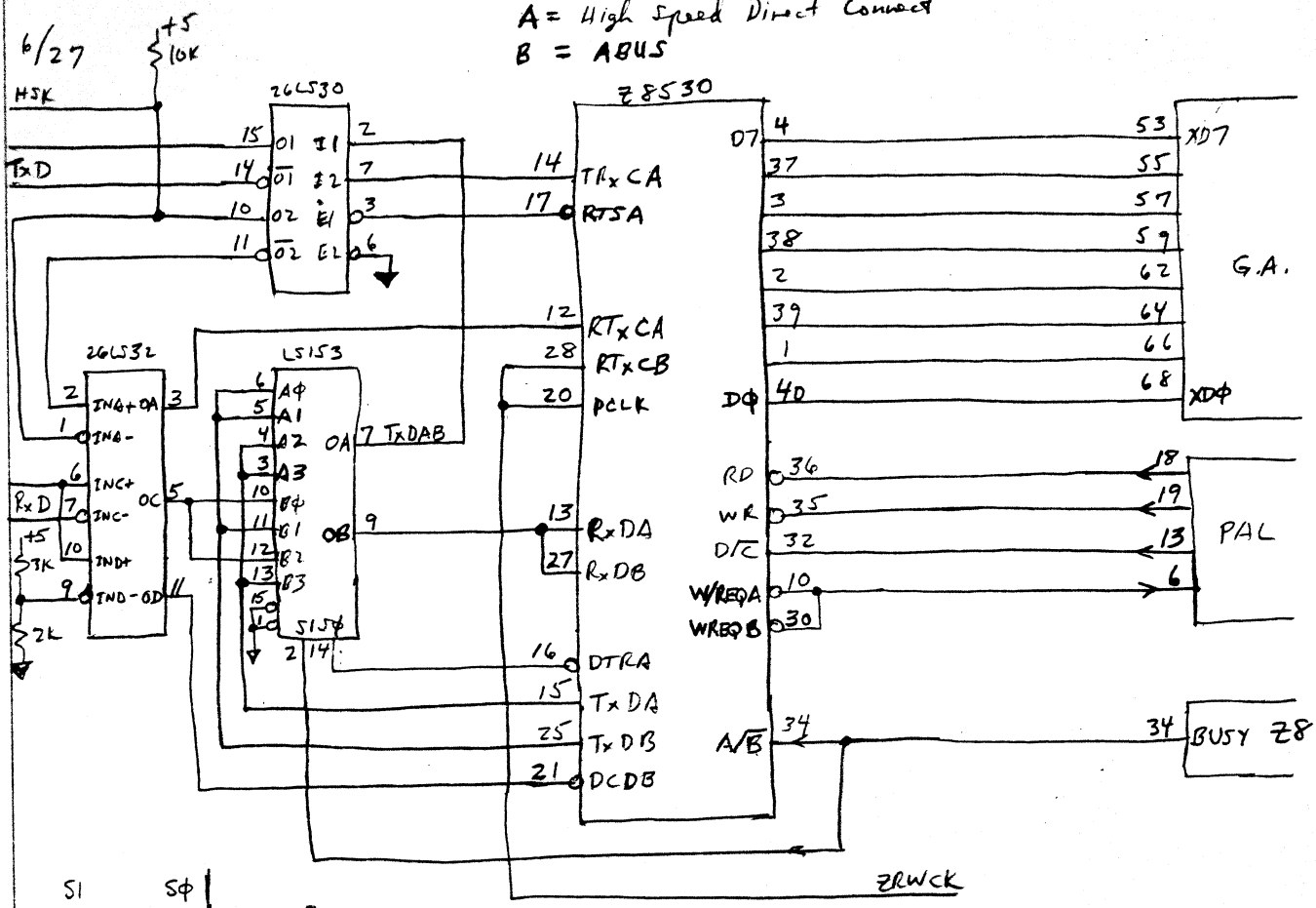
This should work fine since TRxC is delayed by a similar amt to TxD (same D/R chips & cable), and worst case RxD to $\overline{TRxC}$ ↑ setup is 243ns and 0ns is spec.

For **MAC → Widget**:

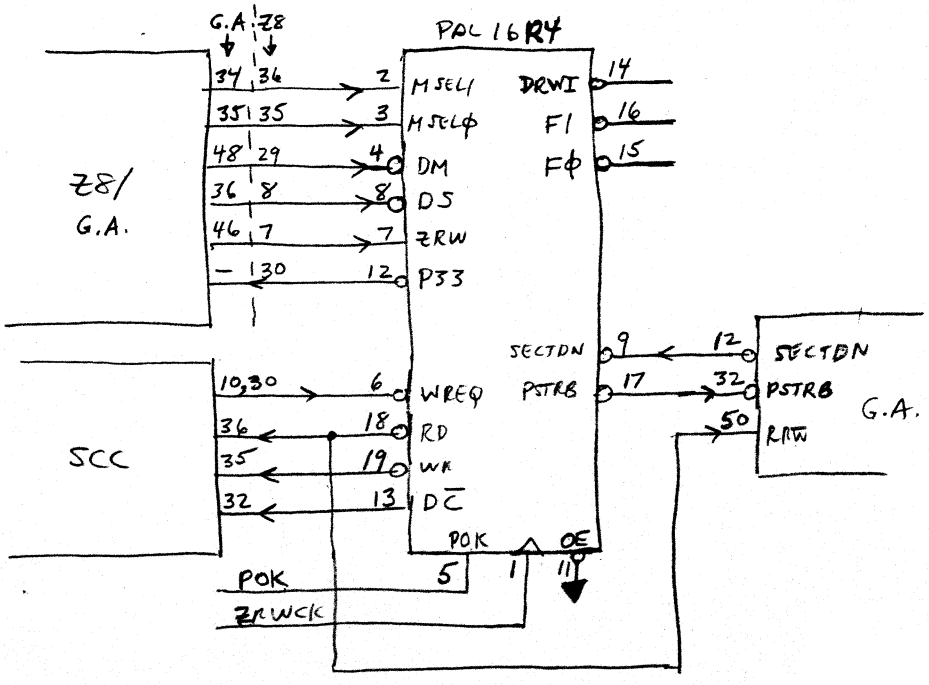


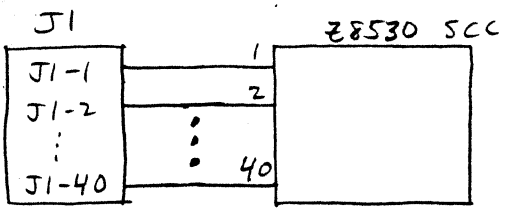
6/27

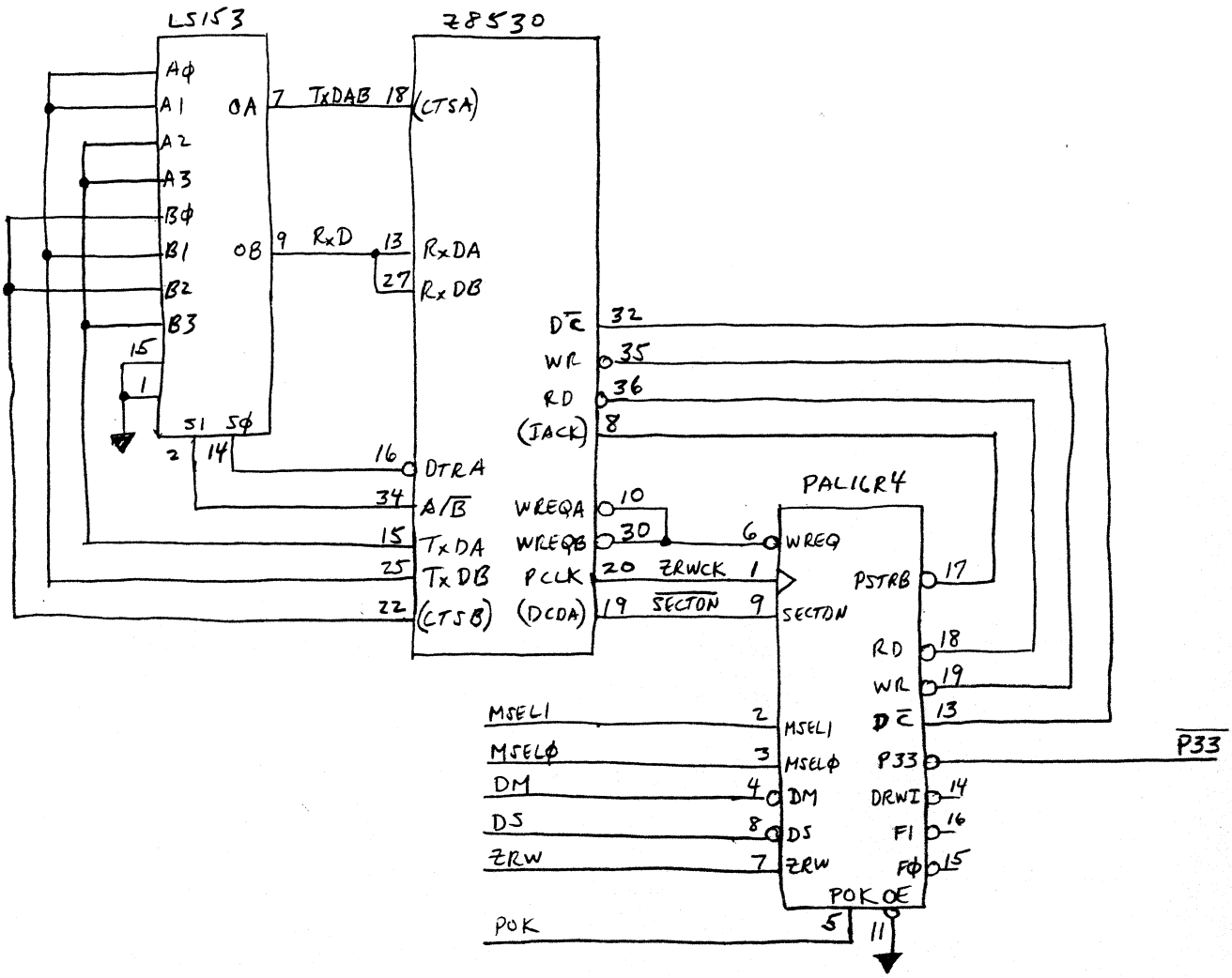
A = High Speed Direct Connect
B = ABUS



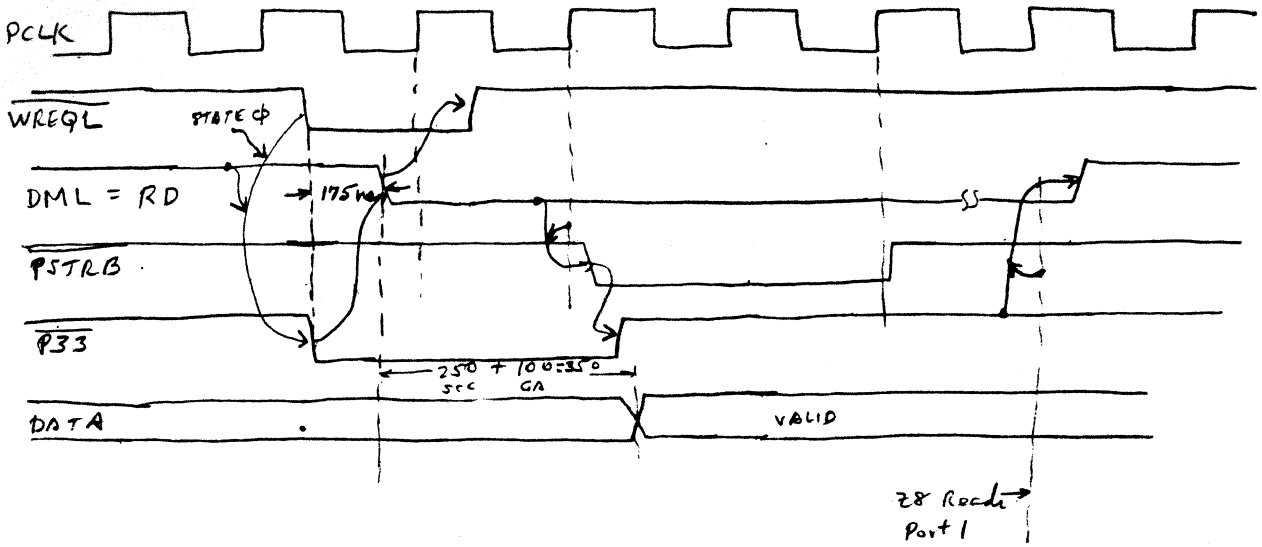
S1 A/B	SØ DTRA	f
0	0	CHB (ABUS) Normal Tx, Rx
0	1	CHB (ABUS) DIAG Tx, Rx
1	0	CHA (HS) Norm Tx, Rx
1	1	CHA (HS) DIAG Tx, Rx



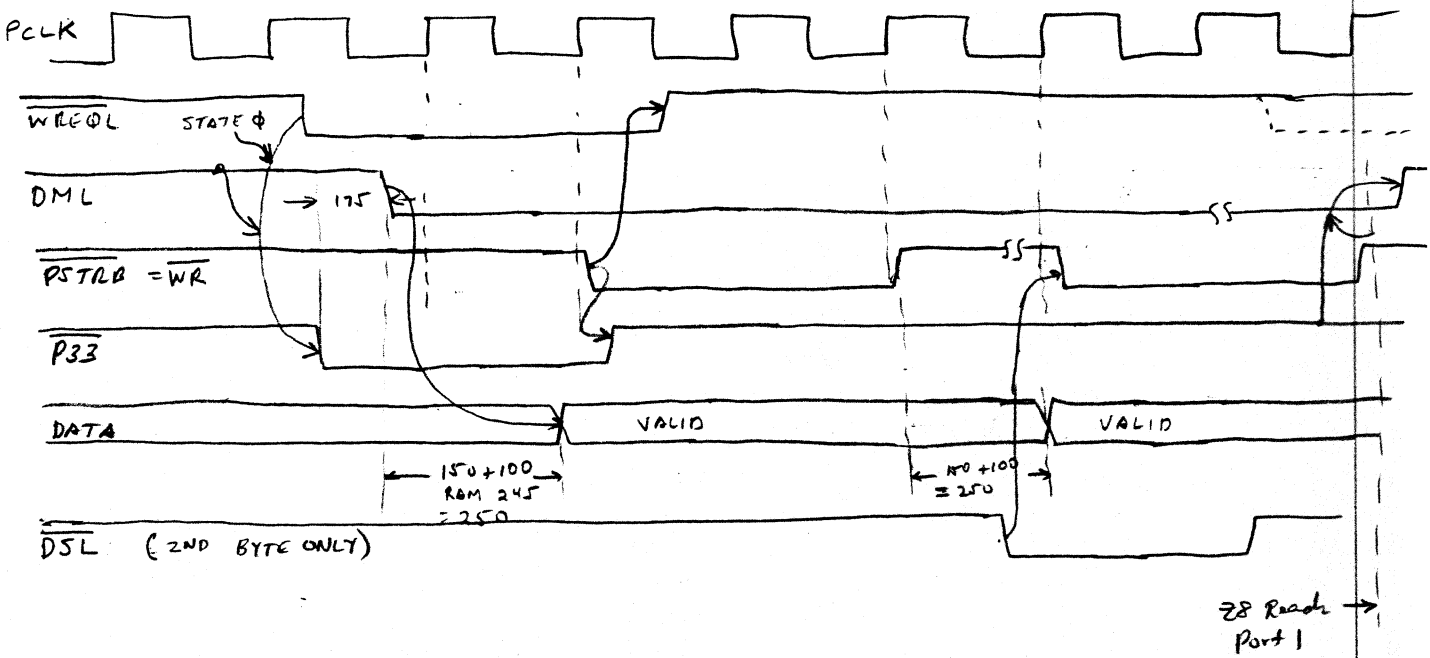

J1
Z8530 SCC
J1-1
J1-2
J1-40
1
2
40


LS153
Z8530
PALCR4
Aϕ
A1
A2
A3
Bϕ
B1
B2
B3
1A
1B
1S1
1Sϕ
TxDAB
RxD
(CTSA)
RxDA
RxDB
DTRA
A/B
TxDA
TxDB
(CTSB)
D/C
WR
RD
(IACK)
WREQA
WREQB
PCLK
(DCDA)
ZRWCK
SECTON
WREQ
PSTRB
MSEL1
MSELϕ
DM
DS
ZRW
POK
OE
P33
DRWI
F1
Fϕ

6/27 READ FROM SCC (DRW = LO):

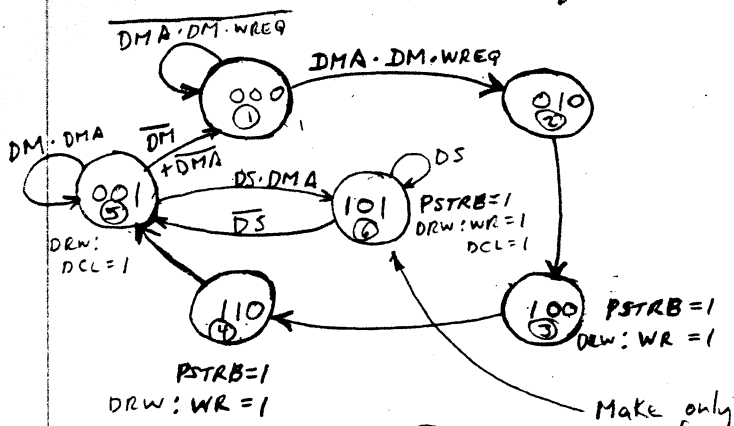


WRITE TO SCC (DRW = HI)



6/25

PAL State Machine Design:



Maps do not include transition from (5) → (6) on DS·DMA

$f_2 f_1$ \ DM WREQ	00	01	11	10
00	1	1	2	1
01	–	–	3	3
11	–	–	5	5
10	–	–	4	4

$f_\phi = 0$

$f_2 f_1$ \ DM WREQ	00	01	11	10
00	1	1	5	5
01	–	–	–	–
11	–	–	–	–
10	–	–	5	5

$f_\phi = 1$

$F2\ F1\ F\phi$

$F2 = PSTRB$

$f_2 f_1$ \ DM WREQ	00	01	11	10
00	000	000	010	000
01	–	–	100	100
11	–	–	001	001
10	–	–	110	110

$f_\phi = \phi$

$f_2 f_1$ \ DM WREQ	00	01	11	10
00	000	000	001	001
01	–	–	–	–
11	–	–	–	–
10	–	–	001	001

$f_\phi = 1$

$$PSTRB = F2 = \overline{f_2} f_1 + f_2 \overline{f_1} \overline{f_\phi} + \overline{f_2} \cdot \overline{f_1} \cdot f_\phi \cdot DS + f_2 \cdot \overline{f_1} \cdot f_\phi \cdot DS$$

$$F1 = \overline{f_1} \cdot \overline{f_\phi} \cdot DM \cdot WREQ + f_2 \overline{f_1} \overline{f_\phi}$$

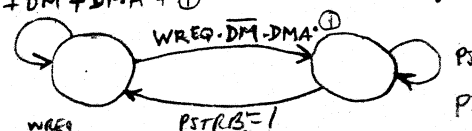
$$F\phi = f_2 f_1 + \overline{f_1} \cdot f_\phi \cdot DM$$

All ANDED w/ POK * DMA

so can reset w/ DMA = φ

P33 SM.: (for scc reads only)

$\overline{WREQ} + DM + \overline{DMA} + \overline{①}$



f \ WREQ DM PSTRB	000	001	010	011	100	101	110	111
0	①	–	①	①	2	–	–	①
1	–	–	②	1	②	–	②	1

f \ WREQ DM PSTRB	00	01	11	10
00	①0	①0	–	②1
01	–	②0	①0	–
11	–	10	10	–
10	–	②1	①1	①1

Need to hold P33 low until PSTRB to guarantee 1 clock cycle width because spec is 175ns min.

$$f = P33 = \overline{DRW} \cdot \overline{f_2} \cdot \overline{f_1} \cdot \overline{f_\phi} \cdot \left(WREQ \cdot \overline{DM} + f \cdot \overline{PSTRB}\right) \cdot DMA$$

← to guarantee ST ⑤ to ST ① transition; i.e. no $\overline{DM}$ $\overline{F2}$ in < 1 clock

[Matches eqn from state dig.]

No race or hazard conditions since all input changes are orderly.

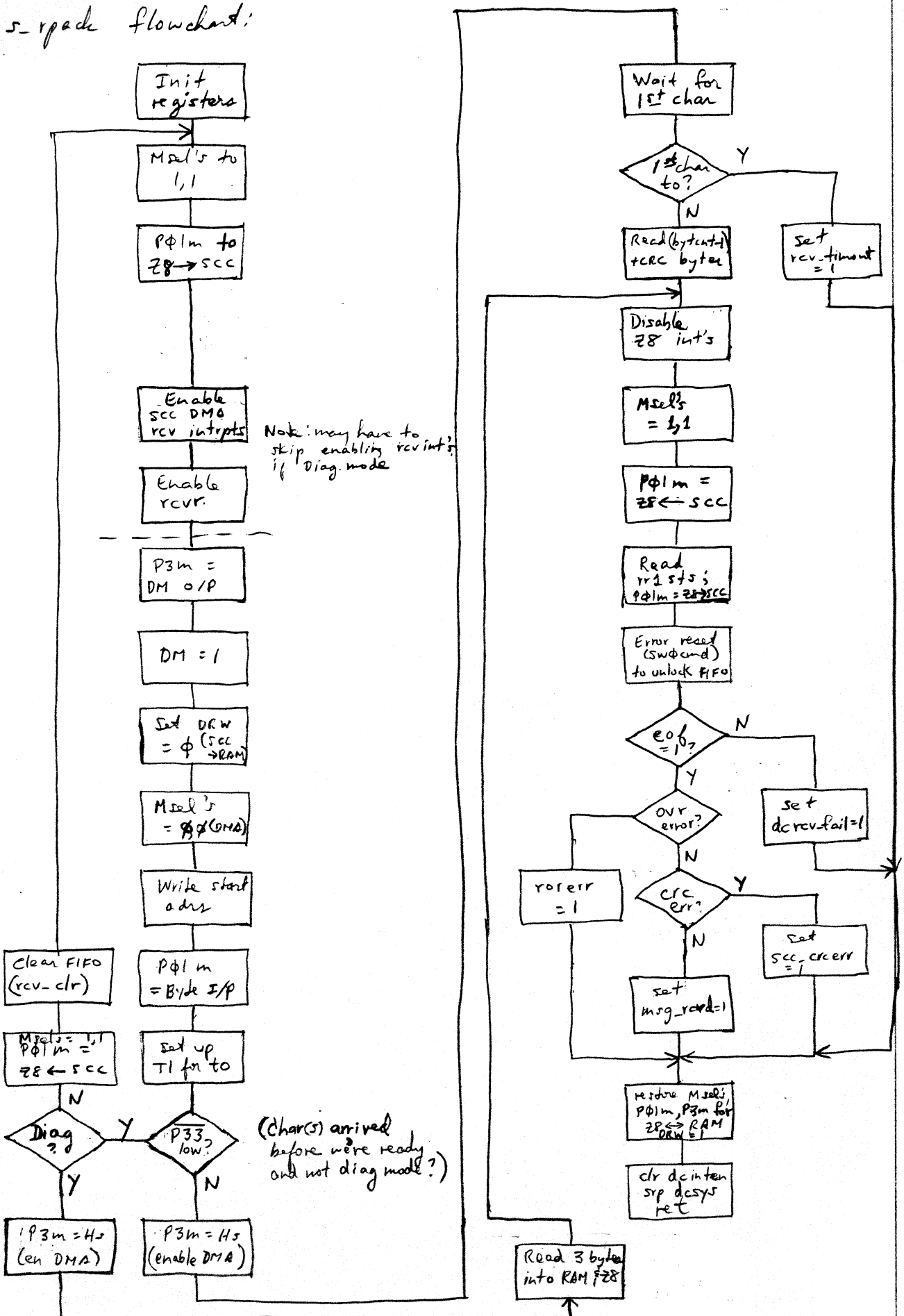
```
PAL16R4
SCCPAL
APPLEBUS
7/31/84 REV0

PCLK MSEL1 MSEL0 DML POK WREQL ZRW DSL SECTDNL GND
/OEL /P33 /DCL /DRWI /F0 /F1 /PSTRB /RD /WR VCC

WR = PSTRB*DRWI + /ZRW*/DSL*MSEL1*MSEL0 + /POK

RD = /DML*/MSEL1*/MSEL0*/DRWI + ZRW*/DSL*MSEL1*MSEL0 + /POK

PSTRB = /PSTRB*F1*POK*/MSEL1*/MSEL0 + PSTRB*/F1*/F0*POK*/MSEL1*/MSEL0 +
        /PSTRB*/F1*F0*/DSL*POK*/MSEL1*/MSEL0 + PSTRB*/F1*F0*/DSL*POK*/MSEL1
        */MSEL0

F1 = /F1*/F0*/DML*/WREQL*/MSEL1*/MSEL0*POK + PSTRB*/F1*/F0*/MSEL1*/MSEL0*POK

F0 = PSTRB*F1*POK*/MSEL1*/MSEL0 + /F1*F0*/DML*POK*/MSEL1*/MSEL0

DRWI = ~~/F1*/F0*/PSTRB*~~ZRW*/DSL*POK*/MSEL1*/MSEL0 + DRWI*DSL*POK

DCL = MSEL1 + MSEL0 + /MSEL1*/MSEL0*DRWI*/F1*F0

P33 = /WREQL*DML*/MSEL1*/MSEL0*POK + P33*/PSTRB*/MSEL1*/MSEL0*POK +
      MSEL1*/SECTDNL*POK + ~~MSEL0*/SECTDDNL*POK~~
```

; Make sure DM is high when ~~changing~~ DRWI - otherwise a DMA cycle will (may) start

PSTRB*/F1*/F0

/MSEL0 + MSEL0/WREQL*POK

hs_rpack flowchart:



8/7

hs_rpack

Receive a high speed packet from host using Z8 handshake lines P33 and P34 (P34 is DM). Data is transferred directly from the SCC chip to RAM via the host DMA channel in the gate array. A hardware state machine implemented in a PAL synchronizes the Z8 handshake signals to the SCC DMA signal (WREQ) and the gate array DMA channel (PSTRB).

INPUTS: 1) rcv buffer pointer in rERF

2) Widget node number stored in SCC

3) timeout value for packet reception in rCrD: # of 2.36 ms time increments

4) lap_only bid in bus_stat 1 for 1 byte reception

OUTPUTS: 1) bus_stat 1: msg_rcvd if packet is received without a timeout
2) bus_stat 3: scc_timout if routine hangs and take a deadman interpt
3) bus_stat 4: rovrerr set if overrun detected,
dc_rcvtail if no eof
rcv_timout if timeout before incoming pkt detected
scc_crcerr if one occurs.

REGISTERS:

r0-r3 : IO Ports
r4 : temp
r5 : temp
r6 : mask for DM active
r7 : used for software timer in wait 1st char loop
r8:
r9:
rA:
rB:

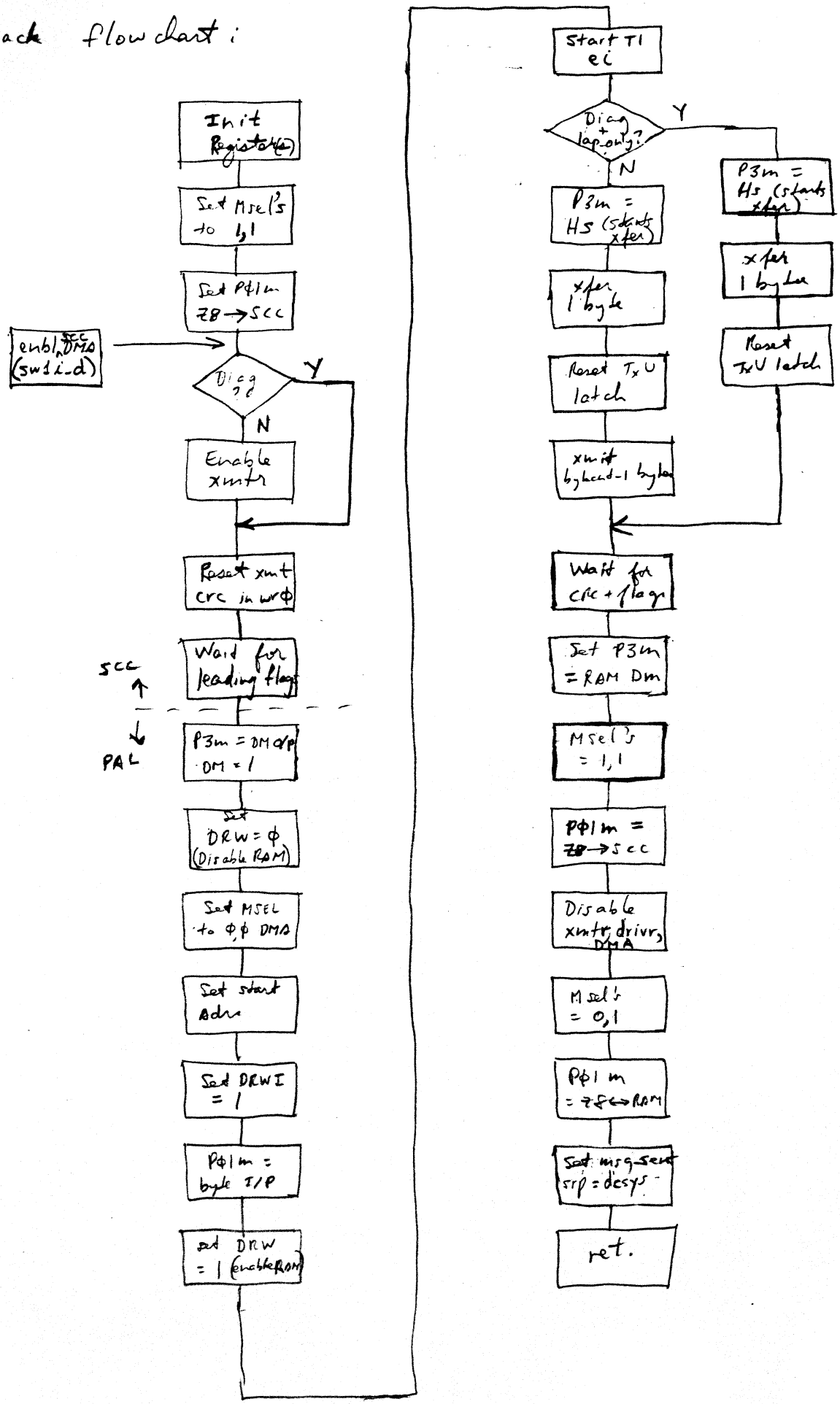
rCrD: timeout value for packet detection
rERF: ptr to rcv buffer in RAM

```
hs_rpack:
; init wrkg registers:
                srp   #wrk_dctr.        ; wrkg reg ptr
                ld    !r6, #Dm          ; Dm mask
                clr   !r7               ; init software timer
hs_rcvstrt:
                or    !r2, #Z8_SCC      ; Msels to 1,1 : Z8 <-> SCC
                ld    P0Im, #P0Im_bo    ; byte o/p
                ld    !r1, #sw1i_d      ; point to sw1
                lde   @!!r4, !r1
; error reset to unlock FIFO & clr status bits
                ld    !r1, #dma_rcv + dma_en + rcv_inten ; enable WREQ for
                                                         ; rcvr, and rcv int on
                                                         ; special cond. (eof)
                lde   @!!r4, !r1
                ld    !r1, #sw3rcv
                lde   @!!r4, !r1
                ld    !r1, #rcven
                lde   @!!r4, !r1          ; enable rcvr.
                ld    P3m, #P3m_Dmio      ; Dm = o/p
                or    !r3, !r6            ; Dm = 1
                and   !r2, #$FF-DrwL_sccW ; set DRW = 0 (SCC -> RAM)
                and   !r2, #$FF-Msel1-Msel0 ; Msels = 0,0 (DMA on)
                ld    !r1, !dmaxbf1o      ; set start adrs low byte
                lde   @!!dmaxbf, !r1      ; P1 = byte o/p ; P0 = A8-11
                ld    P0Im, #P0Im_bi      ; P1 = byte I/P
; set up for T1 dead man timeout
                clr   Pre1
                clr   T1
                ld    Tmr, #Int_Out       ; stop T1
                or    Process_cntrl, #sccinten ; flag for int. routine
; now check for char. already in when we're not ready or diag. mode:
                tm    !r3, #P33L          ; char. already in?
                jr    nz, hs_stdma        ; no. if taken
                tm    !bus_stat1, #seediag ; branch to
                jr    nz, hs_rcvdg        ; diag mode rcv.
; char was already in and not diag mode if get here, so restart rpack
                or    !r2, #Z8_SCC        ; Msels = 1,1
                ld    !r5, #3             ; loop counter to clr FIFO
hs_rcv_clr:     ld    P0Im, #P0Im_bo      ; Z8 -> SCC
                ld    !r1, #sw8buf
                lde   @!!r4, !r1          ; point to SCC rd buffer
                ld    P0Im, #P0Im_bi      ; SCC -> Z8
                lde   !r1, @!!r4          ; read SCC buffer
                djnz  !r5, hs_rcv_clr     ; 3 times
                jr    hs_rcvstrt          ; back to beginning

hs_stdma:
```

hs_rpack (cont.)

```
                ld      P3m, #P3m_Dmhs          ; set hs mode to enable DMA
hs_wistch:
                tm      !r3, !r6                ; Dm low?
                jr      z, hs_got1st
                dec     !r7                     ; bump timer
                jr      nz, hs_wistch
                decw    !!rC                    ; dec large timer
                jr      nz, hs_wistch
                or      bus_stat4, #rcv_timout
                jp      hs_rcv_exit

hs_got1st:
                ld      !r5, !r1                ; complete 1st byte hs
                or      Tmr, #T1_CntEn+T1_Load  ; start 18 ms decd mon tm
                ei

hs_rcvlp:
                ld      !r5, !r1                ; 2nd byte hs & all even bytes
hs_wchan:
                tm      !r3, !r6                ; Dm low?
                jr      nz, hs_wchan            ; wait for 3rd byte (odd bytes)
                ld      !r5, !r1                ; 3rd and all odd byte hs complete
                swap    !r5                     ; timing for blind read
                decw    bytct/2a                ; adjusted for 1st two bytes
                                                ; needs to be (n-2)/2 ; n = # bytes
                                                ; but works for 1 byte rcv @ value = 1
                jr      nz, hs_rcvlp
                di
                or      !r2, #Z8_scc            ; Msels = 1,1
                ld      P01m, #P01m_bo          ; P1 = O/P
                ld      !r1, #sr1sts
                lde     @!!r4, !r1              ; point to sr1
                ld      P01m, #P01m_bi          ; P1 = I/P
                lde     !r1, @!!r4              ; read sr1
                ld      !rC, !r1                ; save status while reset FIFO
                ld      P01m, #P01m_bo          ; back to O/P
                ld      !r1, #errres
                lde     @!!r4, !r1              ; err. reset to clr rcv int & FIFO
                and     !r2, #$FF-Msel1         ; Msels to 0,1; Z8 <-> RAM
                ld      P01m, #P01m_Zbs         ; P1 = ZR bus.
                ld      P3m, #P3m_DmP           ; DM pulse mode for RAM
; now check error conds -> or    !r2, #DrwL_sccW ; DRW = 1
                tm      !rC, #eof               ; check error cond's.
                jr      z, hs_eof
                or      bus_stat3, #dc_rcvfail  ; no eof error
hs_eof:         tm      !rC, #ror
                jr      z, hs_no_ovr
                or      bus_stat3, #rorerr
hs_no_ovr:      tm      !rC, #crc
                jr      z, hs_rcv_exit
                or      bus_stat4, #scc_crcerr
hs_rcv_exit: ---- and   Process_cntrl, #$FF-sccIntEn
                srp     #wrk_dctr
                ret
```

hs_tpack flow chart :



8/6

hs_tpack

Transmits a packet at high speed using Z8 handshake lines P33 and P34 (P34 in DM output). Data is taken directly from RAM and transferred to the SCC via the host DMA channel in the gate array. The handshake lines P33 and P34 synchronize the incoming data to the DMA channel. A diagnostic mode or lap_only transfer will transmit one byte only (plus crc) to implement a fast handshaking protocol between the host and disk.

INPUTS: 1) xmt buffer pointer in rErE

2) bytecount/2 - 2 = bytct/2a: len ^ bytecount divided by 2, adjustment for transmission of 1st pair of bytes, passed in bytecnthi, lo (rA rB) ~~(due to coding of first character transmission). Adjustment~~

3) sccdiag bit in bm_stat1

4) lap_only bit in bm_stat1 to signify a single byte xfr

OUTPUTS: 1) bm_stat1: msg_sent; set if transmission completes

2) bm_stat3: scc_timout set if interrupt before xmsn completes or dead man timer (T1)

REGISTERS:

r0-r3	: IO PORTS	rA rB	: bytct/2a value
r4	: temp	rC	:
r5	: "	rD	:
r6	: Mask for DM active	rErF	: pointer to xmt buffer in RAM
r7	:		
r8	:		
r9	:		

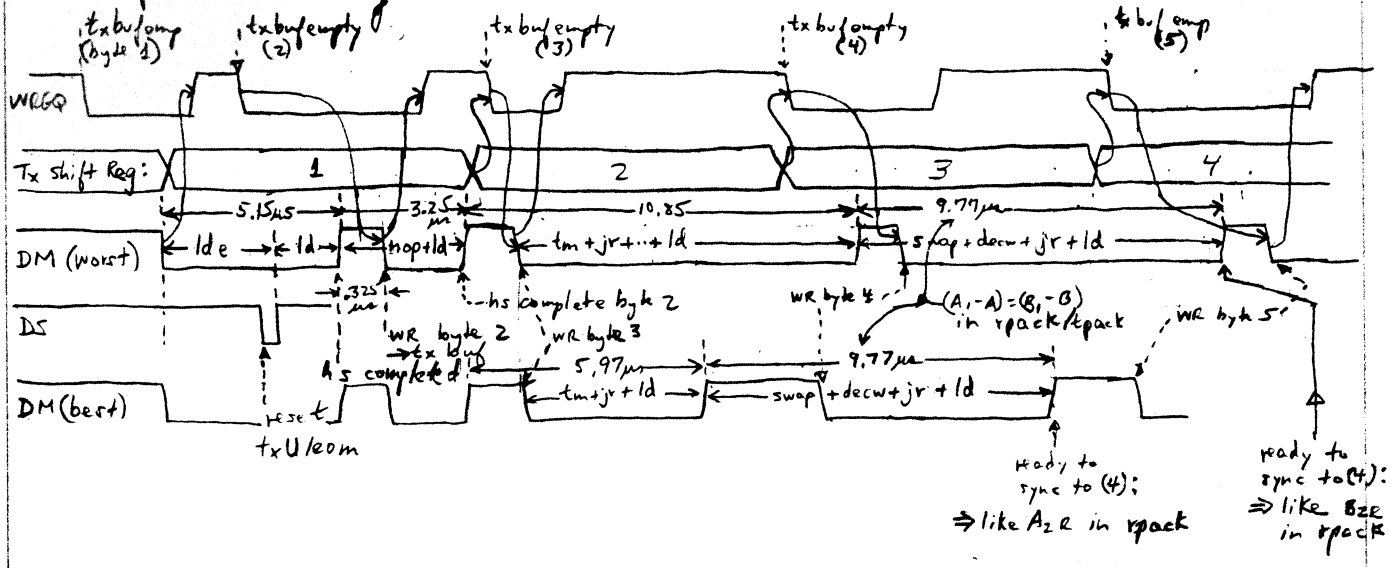
```
hs_tpack:
; init working registers:
                srp   #wrk_dcsys          ; wrky set Φ
                ld    !r6, #Dm            ; setup Dm mask
; set up Ports Φ,1 for Z8 → scc and init scc for xmt
                or    !r2, #Z8_scc        ; Msels = 1,1
                ld    PΦIm, #PΦIm_bo      ; P1 output PΦ-3 adrs 8-11
                ld    !r1, #sw1i_d        ; pt to sw1
                lde   @!!r4, !r1          ; r4r5 are don't cares
                ld    !r1, #dmc_xmt+dma_en ; enable WREQ for xmt
                lde   @!!r4, !r1
; now enable xmtr and RTS for driver if not diag
                ld    !r5, #xmten
                tm    bm_stat1, #sccdiag
                jr    z, hs_en_drivr
                and   !r5, #$FF-RTS-DTR   ; RTS, DTR off for diag.
hs_en_drivr:
                ld    !r1, #sw5xmt
                lde   @!!r4, !r1          ; pt to sw5
                ld    !r1, !r5
                lde   @!!r4, !r1          ; enable xmitr, also RTS, DTR
                                          ; if not DIAG.
                ld    !r1, #rtcrc
                lde   @!!r4, !r1          ; reset tx crc gen in sw Φ
; now setup scc interface logic (PAL16R4 in discrete version) and Z8 mode
; for DMA xfer. Note that flags should now be xmitting on cable.
                ld    P3m, #Dmio          ; Dm = o/p on Port 3
                or    !r3, #Dm            ; set Dm = 1 to inhibit PAL st mach.
                and   !r2, #$FF-DrwL_sccW ; set DRW = Φ temporarily
                                          ; to disable RAM OE to
                                          ; avoid bus fight w/ Z8.
                and   !r2, #$FF-Msel1-MselΦ ; set Msels to Φ, Φ to
                                          ; enable DMA in GA & PAL
                ld    !r1, !dmaxbflo      ; set up to load RAM starting adrs
                lde   @!!dmaxbf, !r1      ; r1 should be a don't care, but
                                          ; set it to adrs lo to be safe
                lde   !r5, @!!r4          ; Do a read to set DRWI = 1
                                          ; in PAL
                ld    PΦIm, #PΦIm_bi      ; set Port 1 to input
                or    !r2, #DrwL_sccW     ; set DRW for RAM → scc:
                                          ; enables RAM onto Z8 bus
; We're now ready to start DMA xfer - only thing remaining is to set P3m
; for handshake mode. DM currently set hi to inhibit PAL State Machine from
; starting a 'write scc' sequence.
; Set up dead man timer with T1:
                clr   Pre1                ; max value
                clr   T1                  ; "
                ld    Tmr, #Int_Out       ; stop T1
                or    Procm_cntrl, #sccinten ; set up flag for interrupt routine
```

hs_tpack (cont.)

```
; Now check for diagnostic or lap_only transmit (both one-byte transmits):
                tm      bus_stat1, #sccdiag+lap_only
                jr      nz, xmt_1byt
                or      Tmr, #T1_CntEn+T1_Load      ; start 18 ms timer
                ei                                  ; enable dead man tmr intrpt.
                ld      P3m, #P3m_Dmhs              ; set handshake mode to
                                                    ; begin DMA xfr - byt 1 out now
13              lde     !r5, @!!r4                  ; Do a read to cause extra
                                                    ; scc write for return byte
                                                    ; in RAM - under SM control
                ld      !r5, !r1                    ; complete hs for byte 1
                nop                                 ; allow time for SM to xfr byt 2
hs_xmtlp:       ld      !r5, !r1    ; complete hs for byt 2 ; done
hs_wbufemp:     tm      !r3, !r6    ; and subsequent even bytes ; in sync w/ (ahead of) tx byte
                jr      nz, hs_wbufemp              ; Dm low?
                ld      !r5, !r1                    ; sync to beginning of hs
                swap    !r5                         ; complete hs for byt 3 & odd bytes
                decw    bytcnt/2a   ; value = (n-2)/2   ; delay for blind write
                ld      !r5, !r1    ; incl. crc     ; bump cntr and delay some more
                                    ; n: # bytes, min=4 ; were past byt 3 → tx SR so
                                                    ; it's safe to complete hs for byt
                jr      nz, hs_xmtlp                ; do it at top of loop
                                                    ; back to sync up to byt 4
hs_xfrend:      di                  ; done w/ xfer so dis. int's.   ; starting out in Tx SR, and so on!
; Now wait for crc and flgs to xmt while setting up to disable xmtr
                ld      P3m, #P3m_Dm P              ; complete SM sync
                or      !r2, #z8_scc                ; Dm → pulsed mode - set it to,
                ld      P01m, #P01m_bo              ; MSel's to 1,1 : z8 ↔ scc; dis DMA
                ld      !r4, #hs_crc_flgs           ; Port 1 to output
                                                    ; wait for crc and flags
hs_wp_end:
                djnz    !r4, hs_wp_end              ; wait for (end of packet)
; turn off xmtr and DMA and driver
                ld      !r1, #sw1i_d
                lde     @!!r4, !r1                  ; point to sw1
                ld      !r1, #dma_init              ; clear WREQ on scc
                lde     @!!r4, !r1
                ld      !r1, #sw5xmt
                lde     @!!r4, !r1                  ; pt to sw5
                ld      !r1, #xmtinit
                lde     @!!r4, !r1                  ; shut off RTS, xmtr
; now restore normal z8 ↔ RAM mode
                and     !r2, #$FF-Msel1             ; set MSel's to 0,1
                ld      P01m, #P01m_zbs             ; z8 ↔ RAM
                or      bus_stat1, #msg_sent        ; set good status
                srp     #wrk_absys
                ret

xmt_1byt:
                ld      P3m, #P3m_Dmhs
                lde     !r5, @!!r4                  ; xfr byte and set return in sw
                jr      hs_xfrend                   ; go to xfr end point to wrap up
                                                    ; single byte xfr
```

Don't forget xmt_1byt

hs_tpack timing:



Looks ok → Always one byte ahead of data going out so tx buffer is full and always syncs up with beginning of byte transmission in tx shift register.

Note that by putting ld after jr in loop, can prevent an erroneous DMA cycle from occurring via the PAL since DM will be set hi under ZP control to complete the last sequence in PAL SM; i.e. allow transition from ⑤ to ①.

7/2

Timing loops:

rpack:

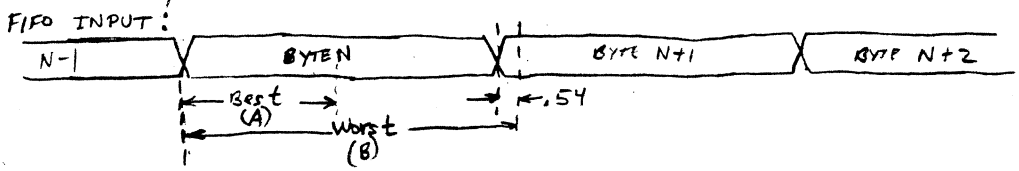
Basic wait_char loop: (bank 0 is working set)

```
wait_char:  6   tm   !r3, !rx      ; rx = $10 for DM
           12/10 jr   nz, wait_char
```

Best case sync time is 16 cycles = 4.34 μs = ½ byte time

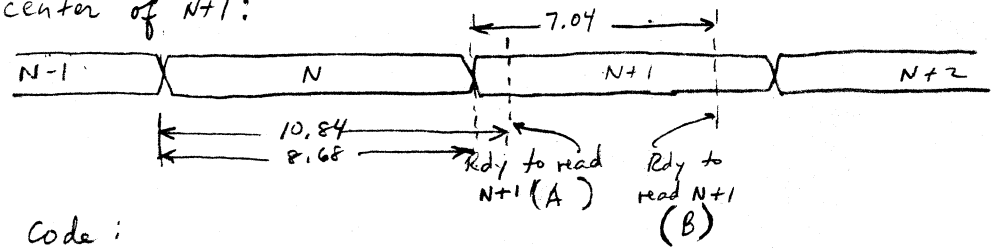
Worst case sync time is 6+12+6+10 = **34 cycles** = 9.22 μs = 1.06 byte time

So sync time looks like:



Since you have a 3-byte FIFO, won't lose BYTE N+1, but want to get in a known place (in time) for blind reads:

Wait slightly > ½ byte time for next read (N+1) so Best Case sync will be guaranteed to read N+1, and Worst Case sync will be approx. in center of N+1:



Code:

```
wait_char:  6   tm   !r3, !rx
           12/10 jr   nz, wait_char
            6   ld   !r5, !r1      ; read P1 ; stores N → RAM
6.5μs       6   nop
            6   nop
            6   nop
```

This gives 2.16 μs of margin for A and 8.68 - 7.04 = 1.64 μs for B.

Could center it better if used two dummy swaps instead of nop's, e.g.

```
5.96        8   swap   !r5
            8   swap   !r5
```

which gives 1.62 μs margin (A) and 8.68 - 6.5 = 2.18 μs for B

So 1st case is better centered by 20 ns (using nop's)

For subsequent blind reads, need to delay a complete byte time to maintain A & B margins, or 8.68 μs = 32.0295 cycles

⇒ use 4 nop's or 6 cycle instructions + swap (8) = 32 cycles.

But only need to do a blind read every other byte:

7/5

According to my calculations, a blind read ratio of ½ would work,

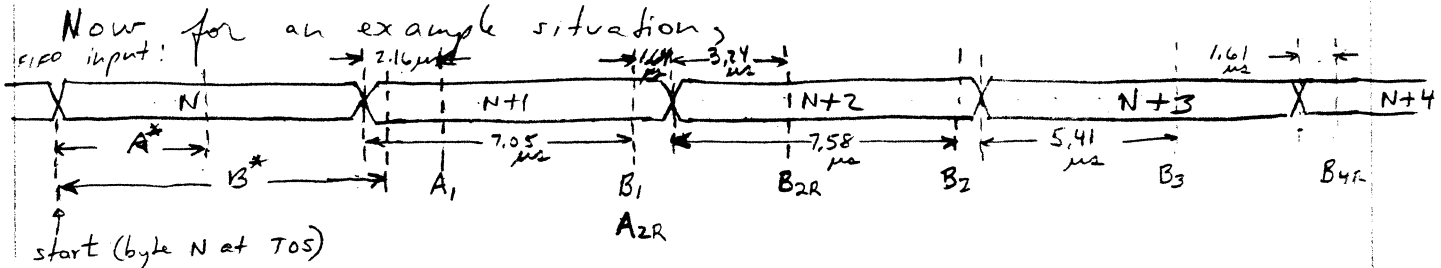
i.e., w_c: 6 tm !r3, !r x
12/10 jr nz, wait_char
6 ld !r5, !r1
10 decw bytecnt/2
12/10 jr nz, w_c

Avg. loop time for ½ is

$$\left[\frac{(6+12+6+10)+(6+10)}{2} + (10+12)\right]\frac{1}{2} + 6$$

= 8.00 μs < 8.68 μs byte time.

Now for an example situation;



A & B are best & worst case sync times, Code for 1/2 is:

```
A2R or B2R - hs_rcvlp:  ld   !r5, !r1      ← *(A = 4.34μs = ½ byte time; B = 9.22μs = 1.06 byte time)
                                            (A1 − A = B1 − B = 24 cycles = 6.51 μs)
             w_c:       tm   !r3, !rx
A or B -------          jr   nz, w_c       ← complete hs for N−1 (even byte), N+1, N+3, etc.
                6  ld   !r5, !r1           ← complete handshake for N (odd byte), N+2, etc.
                8  swap !r5
A1 or B1 -----  10 decw bytecnt/2          ← dummy instr to get 8 cycles.
                6  ld   !r5, !r1           ← ~~complete handshake for N+1~~
                12/10 jr nz, hs_rcvlp      ← Put at top of loop to avoid xfrg an extra byte
```

(disable DM before completing last hs to avoid spurious DMA cycle in PAL)

Note:
A or B = A_2 or B_2 (even); A_1 or B_1 = A_3 or B_3 (odd); etc.

1) Since A_{2R} is reached ~~before~~ byte N+2 arrives, this case is OK.

2) B_{2R} = ready to look for N+2 → syncs up in best case time (A_{2R}), so B_2 is reached within N+2 as shown.

3) B_3 = ready to read N+3; falls in N+3; B_{4R} is reached just after start of N+4.
Since B_{4R} − N+4 (start) < B_{2R} − N+2 (start), looks like we're catching up to the data. But it might be possible to do a blind read at the wrong time--

But it's impossible to do a blind read at the wrong time because that would require B_{2n} (B_2, B_4, etc) to fall between the start of a byte and point (A − 2.16μs), which would force a blind read B_{2n+1} (B_1, B_3, B_5, etc.) of the same byte read at B_{2n}. This is impossible since even bytes are always synchronized and therefore must be read between A and B. The delay A_1−A for the blind read then guarantees that the next byte will be at TOS before the blind read.

Note that scheme fails if FIFO is partly full when you start, because

7/6

B_{2n} could fall before point A in time, and if byte N+1 had not arrived yet, a blind read would be done before it's arrival. So it's necessary to start in sync with data.

Look at rpack loop waiting for 1st byte:

```
7/10  w_1st_c:  tm      !r3, !rx
      12/10     jr      z, got_1st
        10      tm      Irq, # Timer 1
      12/10     jr      z, w_1st_c
        10      and     Irq, #$FF-Timer 1
        10      decw    !rC                 ; rcv timeout counter
      12/10     jr      nz, w_1st_c
                :
              exit
got_1st:  ld    !r5, !r1                    ; complete handshake, byte -> RAM
```

Sync times: (to got_1st point)

Best case time = 18 cycles = 4.88 μs.

Worst case time = 6 + 10 + 10 + 10 + 10 + 10 + 12 + 6 + 12 = 86 = 23.3 μs

= 2.6875 bytes

This means that you wouldn't know where the data started coming in and the blind reads would not work. (see above)

Therefore, need to stay in a tight loop and interrupt out of it, or just sit on the bus and do self tests after every nth command.

tpack:

Have same basic loop as rpack, except 'wait_char' becomes 'wait_bufemp' and need to do one ldc after 1st byte is written to reset TxU/EOM latch:

```
w_be1:          tm      !r3, !rx    } Not needed - just set DMA = 1
A or B ---      jr      nz, w_be1   }
          13    ldc     !r5, @!!r6  ; Do a read to pulse DS, and not
                                    ; affect RAM adr counter
                                    ; complete the handshake for byte 1
           6    ld      !r5, !r1
A1 or B1 ---       6    nop         ; allow time for SM to transfer byte 2
A2n or B2n  hs_xmtlp:  6  ld  !r5, !r1  ; complete hs for byte 2 and subsequent even bytes
  w_be:            6    tm      !r3, !rx    ; note B2n
A or B ---      12/10   jr      nz, w_be
                   6    ld      !r5, !r1    ; just shifts B timing a bit earlier (12 cycles)
                        swap    !r5
A1 or B1 -----          decw    bytecnt/2   ; bytecnt = normal bytecnt - 2
                        ~~ld~~  ~~!r5, !r1~~
                        jr      nz, hs_xmtlp
```

rpack (cont.):

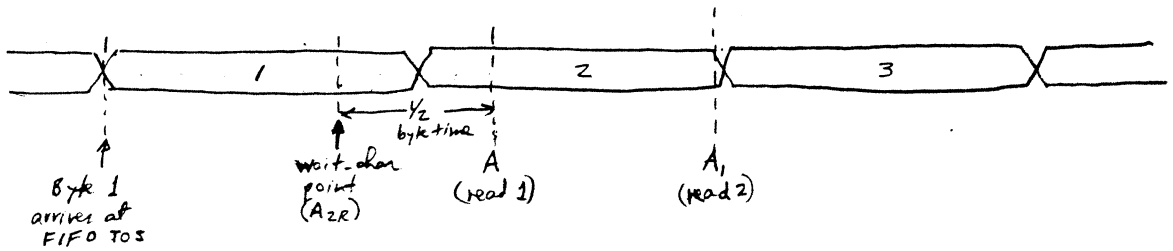
Although don't have time to poll a timer and wait for 1st char, can do a dead man timer interrupt after 1st char is received, just by the 'or Tmr, #T1_CntEn + T1_Load'. Pre1 and T1 can be set before the 'wait_1st char' loop is started; also ei can be set which will also allow for servo interrupts. The 10 cycle 'or Tmr' instruction can replace the 'decw bytecnt/r' and then can fall into the wait_char loop — bytecnt/r then is 1 less since 1st two bytes are read in-line: (compensates for 2 CRC bytes at end - i.e., bytecnt/r is actual bytecnt ÷ 2)

```
            setup
            :
            clr   Pre1          ; set max value
            clr   T1            ; for T1
            ld    Tmr, #Int_Out ; stop T1
            ei                  ; enable int's for rpack & servo
wait_1st char:
            tm    !r3, !rx
            jr    nz, wait_1st char
            ld    !r5, !r1      ; read 1st char
            swap  !r5
            or    Tmr, #T1_CntEn + T1_load   ; start T1 ; 18 ms timeout

hs_rcvlp:   ld    !r5, !r1      ;complete hs for 2nd char and subsequent even chars.
wait_char:  tm    !r3, !rx
            jr    nz, wait_char
            ld    !r5, !r1      ; complete hs for odd chars.
            :
            :
```

Also, eof status (crc, overrun) should be available after last char ei read if interrupt on special condition is used (WR1) since this interrupt locks the FIFO, preserving RR1 status.

Example of blind read failing when not in sync with data (due to a long software timeout loop):

FIFO Input:



But this works since you're reading byte 1 at A, not byte 2 - so 2 doesn't get read twice.

Looks like blind reads will work fine even if not initially in sync with data (ie. FIFO partially filled) because the loop is faster than data rate which means FIFO will get unloaded and eventually the loop will sync up with an incoming byte to an empty FIFO.

However, initial wait_char loop (wait_1stchar) has to be fast enough to not have FIFO overflow in the worst case:

From before, worst case sync time in wait_1st char loop was 23.3μs = 2.6875 bytes < 3 byte FIFO which is OK. However, to simplify the code and still use T1 for dead bus timeout after 1st char is received, use a software timing loop instead of T1 in wait_1st char loop. Then T1 can be set up for the dead bus timeout before the wait_1st char loop is entered and Tmr and ec set to start dead bus timeout after 1st char received as described earlier.

Code follows:

```
        :
clr     Pre 1                   ; set max value
clr     T1                      ; for T1
ld      Tmr, # Int_Out          ; stop T1
or      Proces_cntrl, # sccinten ; set interrupt flag if get a packet
```

7/31 (cont.)

```
wait_1st_char:
          6   tm      !r3, !rX
       12/10  jr      z, got_1st
          6   dec     !rt
       12/10  jr      nz, wait_1st_char
         10   decw    !!rC
       12/10  jr      nz, wait_1st_char
              or      bus_stat4, #rcv_timout
              jr      rcv_exit

got_1st:
              ld      !r5, !r1          ; read 1st byte → RAM
16 cycles instead  10  or      Tmr, #T1_CntEn+T1_Load   ; start 18 ms timer
of normal 18 OK    6  ei
hs_rcvlp:
              ld      !r5, !r1          ; read 2nd byte → RAM
wait_char:
              tm      !r3, !rX
              jr      nz, wait_char
              ld      !r5, !r1          ; read 3rd byte & all odd bytes.
              swap    !r5
              decw    bytecnt/2         ; adjusted for 2 bytes above
              ~~ld      !r5, !r4~~
              jr      nz, hs_rcvlp.
rcv_done:
        [Disable DMA to complete SM sequence and stop further DMA xfrs]
              :
              :
```

Worst case sync time in wait_1st_char loop is

6 + 10 + 6 + 10 + 10 + 12 + 6 + 12 = 72 cycles = 19.512 μs = 2.248 bytes OK

Timeout loop using !rt is 6 + 10 + 6 + 12 = 34 cycles = 9.22 μs

So for rt = 0, get 2.36 ms timeout. Max timeout of loop is 2^{16} (!!rC) * 2.36 = 154 sec. so this is very adequate.

7/31 (cont)

Other new equates -

```
bytct/2a        .equ  $ØX     ; register set Ø bytecnt ÷ 2 variable,
                              ;  adjusted for transmission of 1st 2 bytes
wrk_dctr        .equ  $ØØ     ; working set for direct connect.
PØlm_bo         .equ  PØ_Ø3_Adr + PØ_47_Out + Stack_In + P1_Out
                      + Mem_Ext   ; P1 output, PØ Address, extended mem trng
PØlm_zbs        .equ  PØ_Ø3_Adr + PØ_47_Out + Stack_In + P1_Adr   (used for Z8→SCC)
                              ; P1 adr/data, normal mem trng.
PØlm_bi         .equ  PØ_Ø3_Adr + PØ_47_Out + Stack_In + P1_In
                      + Mem_Ext   ; P1 input, PØ Address, ext. mem trng,
                              ; (used for Z8 ← SCC and SCC DMA xfer)
P3m_DmP         .equ  BØ_7_Ser + B1_6_Io + B2_5_Io + B3_4_Idm
                      + Totem_Pol + Par_Off
                              ; P3m for Z8 ⇔ RAM (Dm = Pulsed)
P3m_Dmio        .equ  BØ_7_Ser + B1_6_Io + B2_5_Io + B3_4_Io
                      + Totem_Pol + Par_Off
                              ; P3m for Dm = 1 or Ø
P3m_Dmhs        .equ  BØ_7_Ser + B1_6_Io + B2_5_Io + B3_4_Hs
                      + Totem_Pol + Par_Off
                              ; P3m for Dm and P33 handshake
```

Note: change all working register equates to wrkg set Ø.

```
hs_crc_flgs     .equ  $Ø8     ; constant for waiting for crc + flgs
                              ; at end of hs_tpack
```

8/2

Appletalk timing using 8530:

tpack: ok since can reset TxU latch after 1st byte is written without changing the pointer bits (wr0 automatically selected after any read or write)

rpack:
Current code -

```
wait_char:      lde   !rC, @!!scc_sts0
                tm    !rC, !w4
                jr    z, wait_char

char_rdy:
                lde   !rD, @!!scc_stat1
                lde   !r0, @!!scc_rbuf
                lde   @!!rcvbuf, !r0
                incw  rcvbuf
                tm    !rD, !r5
                jr    z, wait_char
```

Code modified for 8530 - use P33 to get char_rdy status

```
wait_char:   10  tm    Port3, #P33
             12/10 jr    nz, wait_char

char_rdy:
             13  lde   @!!rcontrol, !rx     ; D/C low, wr0 default, rx = 01 point to rr1
             13  lde   !rD, @!!rcontrol     ; rcontrol forces D/C lo - read rr1 sts
             13  lde   !r0, @ scc_rbuf      ; get data
             13  lde   @!!rcvbuf, !r0
             10  incw  rcvbuf
              6  tm    !rD, !r5
             12/10 jr    z, wait_char
```

Sync time best case = 20 cycles ; worst case = 10 + 12 + 10 + 10 = 42

Procm time = 80 cycles

Best case loop time = 100 cycles = 27.13 μs < 34.37 μs min byte time
Worst " " " = 122 " = 33.1 μs < 34.37 μs " " "

So will work.

Changes are that need $\overline{WREQ}$ line returned to ZP on P33 when MSEL1, 0 are in ZPSEL mode (0, 1), and only return $\overline{SECTON}$ when MSEL1, 0 = SEQSEL = (1, 0)

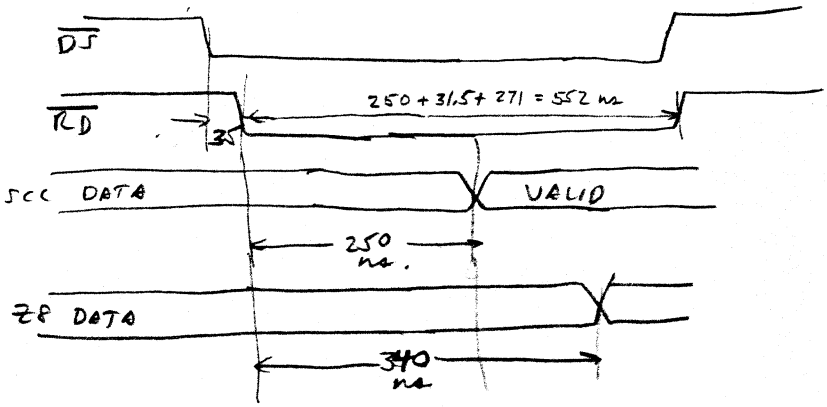
Connect 8530 D7-Dϕ to XD7-XDϕ & PSTRB & RRW (= $\overline{RD}$ to SCC)

Use MSEL2, ϕ = 1,1 which is Z8 ↔ Apple code for programming the SCC. Noting that w245 driver is enabled for 1,1 and direction determined by RRW, and that address counters are loaded for writes regardless of MSEL lines, can generate $\overline{SCC\ SEL}$ via address lines w/ I/O/$\overline{MEM}$ high and $\overline{DM}$ low. For reads from SCC, can set up $\overline{SCC\ SEL}$ address and just do reads w/ $\overline{C/D}$ low or high; $\overline{C/D}$ = $\overline{DMA*}$ = $\overline{DTRA*}$

$\overline{C/D}$ = $\overline{DMA*}$ · ZRW

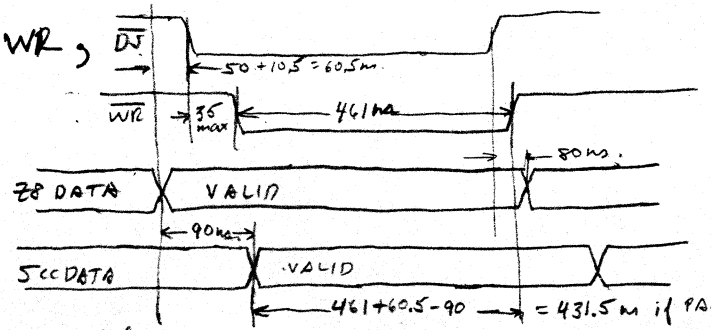
Access time thru Gate Array buffer: 50ns typical, 90ns w.c.

For RD,



From Z8611 spec $t_{dDS(DI)\,min}$ = 200ns for 8MHz clock & no extended timing. With 7.37 MHz clock, change in 3 clock periods + 1 SCLK time are added = (0.5m)3 + 271m = 302.5m.

So $t_{dDS(DI)}$ = 200 + 302.5 = 502.5m > 375 ns OK

For WR, 

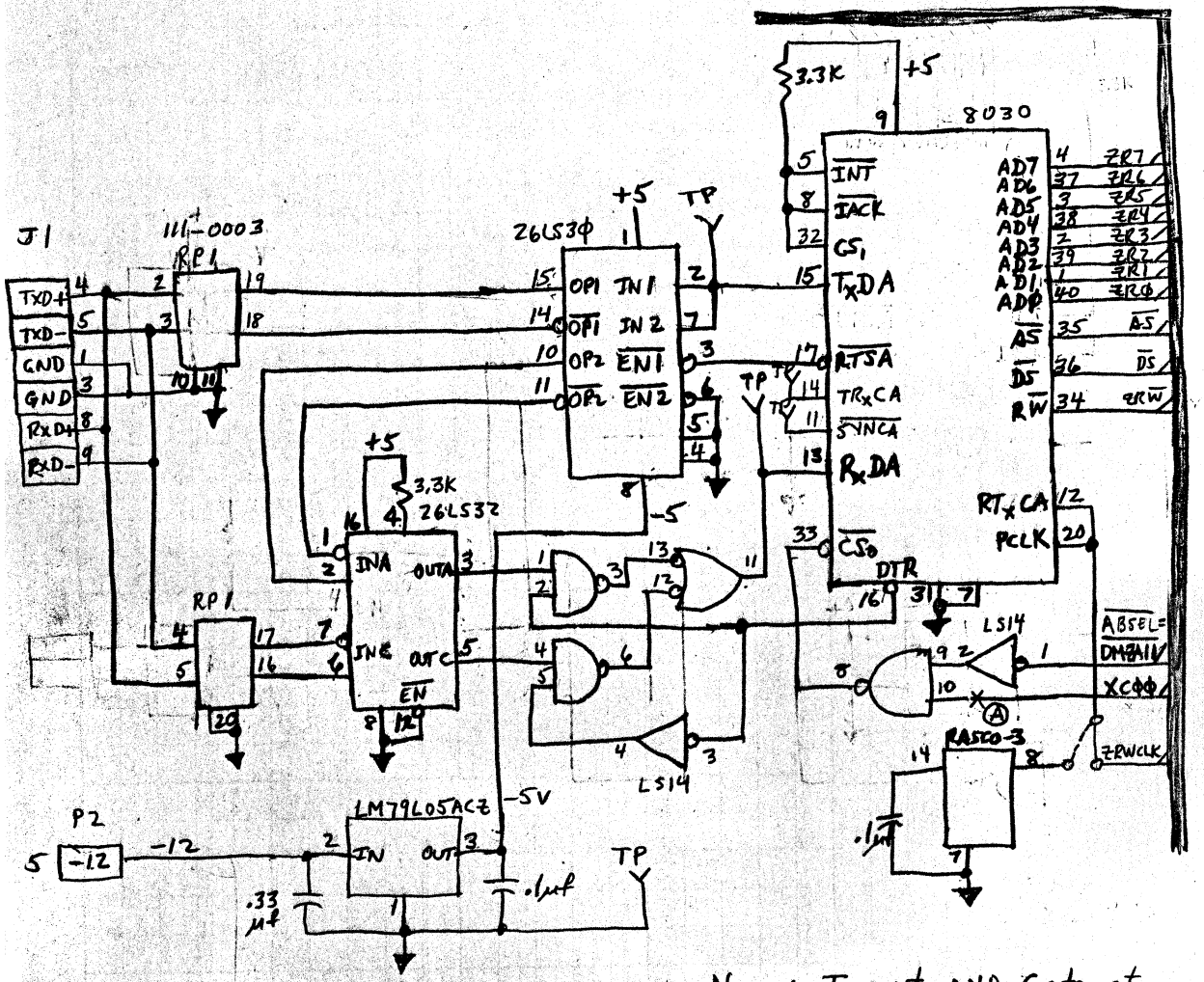
Check Data setup to WR↓ w/ Zilog

WR low width = 160 + 3(10.5) + 271 = 461 ns > 390 ns spec on 8530 OK

However WR Data to WR↓ set-up Time of 0ns in 8530 spec is not met since 90 > 60.5 + PAL $t_{pd\,min}$

Should be OK if SCC data is latched on WR↑ and >390m overlap is guaranteed, which is.

But SCC Data latched on WR↓ so Ide alone won't work,



NOTE ON ZRWCLK:

Current code manually clocks ZRWCLK with ZTEST active (START inactive) to reset state machine if it gets hung. This would require resetting SCC since its clock would effectively be turned off (500KHz min freq). But since ZRWCLK is running at 3.6864 MHz, just set ZTEST active with START high to reset state machine.

IC's: (added)

1 40 PIN
4 14-20 PIN
1 3 PIN (REGULATOR)

J1:

DB-9 rt-angle

APPLE BUS INTERFACE