```
        ;Subroutine IO.RMQ.IDLE, Level 2

        ;On call, BASE=IOOS, L6.DCB points to DCB to be removed from IDLE
        ;heap. DCB.STATE must be STATE.IDLE and it's queue header must match
        ;L3.IDLH.

        io.rmq.idle:
023451   l6.dcb + dcb.state -> mar(r)    ;read the DCB's state
023452   nop                             ;[1]
023453   state.all -> g0.temp            ;[2] make a mask
023454   g0.temp & mbr -> g0.temp
023455   g0.temp - state.idle ls         ;make sure we're in IDLE
023456   ifnot zero crash -> upc         ;if not in IDLE, die
023457   ifnot zero ioerr.rmqnis -> temp

        ;The state is right, let's go check the queue header.

023460   l6.dcb + q.hedr -> mar(r)
023461   l6.dcb -> temp                  ;[1] for Q.RMQ, below
023462   temp -> g0.temp                 ;[2]
023463   117.idlh - mbr ls               ;is it on the IDLE queue?
023464   ifnot zero crash -> upc
023465   ifnot zero ioerr.rmqniq -> temp

        ;The state and the queue header are all okay, call RMQ.

023466   q.rmq -> upc                    ;remove it from the IDLE heap
023467   (.+1) -> l1.ret1

023470   l2.ret2 -> upc                  ;return
023471   nop
```

```
        ;Subroutine IO.ENQ.IDLE, Level 2

        ;Call with DCB in L6.DCB, merely enqueue's it to the IDLE
        ;heap and sets its state accordingly.

        io.enq.idle:
 023472   l6.dcb + dcb.state -> mar(r)    ;change the state (read whole word fi
 023473   l6.dcb -> temp                  ;[1] set up for ENQ call below
 023474   temp -> g0.temp                 ;[2]

 023475   mbr -> g2.temp
 023476   g2.temp & ~(state.ready ! state.timing ! state.pending) -> g2.temp
 023477   g2.temp ! state.idle -> mbr

 023500   l6.dcb + dcb.state -> mar(w)    ;write the new state back
 023501   l17.idlh -> temp                ;[1] more ENQ set up
 023502   temp -> g1.temp                 ;[2]

 023503   q.enq -> upc                    ;put it on end of IDLE queue
 023504   (.+1) -> l1.ret1

 023505   l2.ret2 -> upc                  ;all done
 023506   nop
```

```
        ;Subroutine IO.RMQ.READY, Level 2

        ;Verify and remove a DCB (in L6.DCB) from the READY queue.

        io.rmq.ready:
 023507  l6.dcb + dcb.state -> mar(r)    ;make sure it's on the ready queue
 023510  l6.dcb -> temp                  ;[1] copy DCB address into G0 for DEQ
 023511  temp -> g0.temp                 ;[2]

 023512  mbr -> g2.temp
 023513  g2.temp & (state.all) -> g2.temp
 023514  g2.temp - state.ready ls        ;in READY?
 023515  ifnot zero crash -> upc
 023516  ifnot zero ioerr.rmqnrs -> temp ;note in READY state

        ;State is okay, verify the queue header

 023517  l6.dcb + q.hedr -> mar(r)       ;look at the queue header
 023520  nop                            ;[1]
 023521  nop                            ;[2]
 023522  l4.rdyh - mbr ls                ;on the READY heap?
 023523  ifnot zero crash -> upc
 023524  ifnot zero ioerr.rmqnrq -> temp

        ;Everything is okay, DCB address is in G0 (from above), call
        ;RMQ.

 023525  q.rmq -> upc                    ;remove it
 023526  (.+1) -> l1.ret1

 023527  l2.ret2 -> upc
 023530  nop
```

```
        ;Subroutine IO.ENQ.READY, Level 2

        ;Adds a DCB to the READY heap, adjusting its STATE accordingly.

        io.enq.ready:
023531   l6.dcb + dcb.state -> mar(r)    ;must read whole word to change part
023532   l6.dcb -> temp                  ;[1] set up for ENQ call below
023533   temp -> g0.temp                 ;[2]

023534   mbr -> g2.temp
023535   g2.temp & ~state.all -> g2.temp
023536   g2.temp ! state.ready -> mbr
023537   l6.dcb + dcb.state -> mar(w)    ;write new one back
023540   l4.rdyh -> temp                 ;[1] set pu for ENQ
023541   temp -> g1.temp                 ;[2]

023542   q.enq -> upc                    ;enq it
023543   (.+1) -> l1.ret1

023544   l2.ret2 -> upc
023545   nop
```

```
        ;Timing Insertion/Deletion.

        ;Subroutine IO.RMQ.TIMING, level 2

        ;Enter with L6.DCB = Dcb to be removed.

        io.rmq.timing:
023546   l6.dcb + dcb.state -> mar(r)    ;make sure it's okay to remove it
023547   nop                             ;[1]
02355C   state.all -> g2.temp            ;[2]
023551   g2.temp & mbr -> g2.temp        ;look only at state
023552   g2.temp - state.timing ls       ;in timing?
023553   ifnot zero crash -> upc         ;if not, complain
023554   ifnot zero ioerr.rmqnts -> temp

        ;Right state, now check queue header

023555   l6.dcb + q.hedr -> mar(r)
023556   l6.dcb -> temp                  ;[1] for RMQ, below
023557   temp -> g0.temp
02356C   l5.timh - mbr ls                ;is this really on the timing heap?
023561   ifnot zero crash -> upc
023562   ifnot zero ioerr.rmqntq -> temp

023563   q.rmq -> upc                    ;remove it
023564   (.+1) -> l1.ret1

023565   l2.ret2 -> upc                  ;return
023566   nop
```

```
        ;Subroutine IO.ENQ.TIMING, Level 2

        ;Enter with L6.DCB pointing to the DCB to be inserted. It's wakeup
        ;time, etc. in DCB.TIME.

        io.enq.timing:
023567   l6.dcb + dcb.time -> mar(r)     ;look at time
023570   l6.dcb -> temp                  ;[1] pick up l6.dcb for possible copy
023571   nop                             ;[2]

023572   l15.time - mbr ls               ;compare wakeup time with current tim
023573   ifnot neg io.enq.pending -> upc ;other window, just run him again

        ;Here when the time in MBR is in the future. Save it in L4.TITIME and
        ;save l6.dcb (cached in TEMP) in L5.TITHIS in the TID register block.
        ;Also copy over our caller's address to TID because interrupts will
        ;smash the IOCS rb. "Prime" the insertion loop by setting up TINEXT
        ;to be the first thing in the timing heap.

023574   if neg l5.timh + q.forw -> mar(r) ;start the fetch of the forward pt
023575   rb.tid -> base                  ;[1] change base
023576   mbr -> l4.titime                ;[2] copy MBR (DCB.TIME) before it ch
023577   temp -> l5.tithis               ;set up THIS
023600   mbr -> l0.tinext                ;set up NEXT

023601   rb.ioos -> base                 ;also copy over caller's stack (l3, l
023602   l2.ret2 -> mbr                  ;using MBR and TEMP
023603   l3.ret3 -> temp                 ;as caches
023604   rb.tid -> base
023605   mbr -> l2.ret2
023606   temp -> l3.ret3
                                         ;fall into TI.TOP
```

```
                                        ;Fall into here from previous page, t

        ;This is the top of the timing insert loop. Enter here with THIS, NEX
        ;TIME set up in the TID rb. First, check to see if this is the end of
        ;the timing heap. Then, check to see if this is the right place for t
        ;DCB to be inserted. Then, CDR to the next entry in the heap and (pos
        ;debreak for interrupts.

        ti.top:
023607   rb.iocs-> base                 ;take a look at the other base
023610   15.timh -> mir                 ;to find out where the timing heap st
023611   115.time -> mbr                ;*** hack ***
023612   mbr -> g0.temp
023613   rb.tid -> base                 ;(faster than going to macromemory)
023614   10.tinext - mir ls             ;at header?

        ti.loop:
023615   if zero ti.here -> upc         ;if at "end", insert here

        ;We're not at end. Check to see if the time of this DCB is after the
        ;of the to-be-inserted one (time in TITIME). If so, this is the right
        ;place for the insertion.

023616   ifnot zero 10.tinext + dcb.time -> mar(r) ;read the time of this one
023617   nop                            ;[1]
023620   nop                            ;[2]
023621   10.tinext + q.forw -> mar(r)   ;start read of next in case we need i
023622   mbr -> temp                    ;[1] snatch this one's time first
023623   g0.temp - temp ls              ;[2] skip over old entries
023624   if neg 14.titime - temp ls     ;compare entries
023625   if neg ti.here -> upc          ;if negative, its newer
023626   ifnot neg mbr -> 10.tinext     ;otherwise officially CDR
023627   ifnot intp ti.loop -> upc      ;if no pending interrupts, continue w
023630   10.tinext - mir ls             ;and do TI.LOOPs first check for it

        ;Here when we need to debreak for an interrupt. Enter INT.INSTR with
        ;"return" address in TEMP.

        ti.int:
023631   ti.top -> temp                 ;set up return
023632   int.instr -> upc               ;process interrupts from instruction
023633   rb.softint -> base             ;with correct BASE
```

```
                ;Here from insertion loop with DCB at TINEXT (on timing heap) is
                ;older than one in TITHIS/TITIME. First, switch back to the normal
                ;rb, resoting L6.DCB, etc.

                ti.here:
023634           12.ret2 -> temp                ;using TEMP and MBR, restore caller s
023635           13.ret3 -> mbr
023636           rb.icos -> base
023637           temp -> 12.ret2
02364C           mbr -> 13.ret3
023641           rb.tid -> base                 ;then restore DCB
023642           15.tithis -> temp
023643           10.tinext -> mbr               ;and tinext
023644           0 -> 10.tinext                 ;all done with interlock..
023645           rb.icos -> base
023646           temp -> 16.dcb
023647           mbr -> g1.temp                 ;but put next in G1 for ENQ, below
02365C           temp -> g0.temp                ;and this in G0

                ;Without modifying G0 and G1, update the DCB.STATE to reflect the DCB
                ;new home on the timing heap.

023651           16.dcb + dcb.state -> mar(r)   ;read the old state word
023652           nop                            ;[1]
023653           ~( state.all ! state.retimed ! state.repoked) -> g2.temp ;[2]
023654           g2.temp & mbr -> g2.temp
023655           g2.temp ! state.timing -> mbr
023656           16.dcb + dcb.state -> mar(w)   ;put it back
023657           q.enq -> upc                   ;[1] put it here in the timing heap
02366C           (.+1) -> 11.ret1               ;[2]
023661           12.ret2 -> upc                 ;return to caller
023662           nop                            ;[1]
```

```
        ;IO.RMQ.PENDING, level 1

        ;Subroutine to remove a DCB (in L6.DCB) from its PENDING queue.
        ;If we are removing the last remaining DCB from the queue, we
        ;turn off the system attention mask for that priority.

        io.rmq.pending:
023663   l6.dcb + dcb.state -> mar(r)    ;make sure it's in the right state
023664   priority.all -> g0.temp         ;[1]
023665   state.all -> g2.temp            ;[2]
023666   g2.temp & mbr -> g2.temp
023667   g2.temp - (state.pending) ls    ;in a pending state?
023670   ifnot zero crash -> upc
023671   ifnot zero icerr.rmqnps -> temp ;if not, bomb

        ;State looks right. Now find the queue header for this interrupt
        ;priority by calling IO.FIND.PHEDR, then compare this DCB's Q.HEDR
        ;to it.. MBR still has state/priority word.


023672   g0.temp & mbr -> g0.temp        ;priority only, please
023673   io.find.phedr -> upc
023674   (.+1) -> l1.ret1

023675   l6.dcb + q.hedr -> mar(r)       ;go fetch the dcb's reader
023676   l6.dcb -> temp                  ;[1] copy dcb address to G0 for RMQ c
023677   temp -> g0.temp                 ;[2]
023700   g1.temp - mbr ls                ;a match with FIND.PEDHR's result?
023701   ifnot zero crash -> upc
023702   ifnot zero icerr.rmqnpq -> temp

023703   q.rmq -> upc                    ;remove it
023704   (.+1) -> l1.ret1

        ;Is pending queue now empty? If so, clear ATTN bit

023705   mir ls
023706   ifnot zero l2.ret2 -> upc       ;if not, return okay

        ;Here when we need to turn off the ATTN bit for the priority of this
        ;DCB.

023707   if zero l6.dcb + dcb.priority -> mar(r) ;read the priority again
023710   17 -> g1.temp                   ;[1]
023711   20 -> g2.temp                   ;[2]
023712   g2.temp & mbr ls                ;look at msb of priority for later

023713   g1.temp & mbr -> g1.temp        ;only 4 bits worth
023714   g1.temp + rb.bits -> base
023715   10 -> temp                      ;get mask to turn off bit
023716   rb.ioos -> base                 ;fix base

023717   if zero l10.attn1 ? temp -> l10.attn1
023720   l2.ret2 -> upc                  ;return
023721   ifnot zero l11.attn2 ? temp -> l11.attn2
```

```
        ;IO.ENQ.PENDING, level 2

        ;Subroutine to ENQ the current DCB (in L6.DCB) onto the right
        ;PENDING queue.  Clears all state flags if you enter at IO.ENQ.PENDI
        ;clears all but STATE.REPOKED if enter at IO.ENQ.PEND2

        io.enq.pend2:
023722   l6.dcb + dcb.state -> mar(r)
023723   io.enq.pend0 -> upc                ;[1]
023724   ( ~( state.all ! state.retimed ) ) -> g0.temp ;[2]

        io.enq.pending:
023725   l6.dcb + dcb.state -> mar(r)   ;read old state word
023726   ( ~( state.all ! state.retimed ! state.repoked) ) -> g0.temp ;[1]

        io.enq.pend0:
023727   enq.pending.ret1 -> l1.ret1    ;[2] save a cycle later
023730   g0.temp & mbr -> g0.temp
023731   g0.temp ! state.pending -> mbr ;put us in pending state
023732   l6.dcb + dcb.state -> mar(w)
023733   io.find.phedr -> upc           ;[1] find pending header (RET1 alread
023734   g0.temp & priority.all -> g0.temp ;[2] get priority mask

        enq.pending.ret1:              ;for saving a cycle, above
023735   l6.dcb -> temp                ;set up G0, as well as G1 for ENQ
023736   temp -> g0.temp               ;(G1 set up by FIND.PHEDR)
023737   q.enq -> upc                  ;put this on some pending header
023740   (.+1) -> l1.ret1

023741   g3.rone - mir ls              ;does queue now have exactly on eleme
023742   ifnot zero l2.ret2 -> upc     ;nope, return (all done)

        ;Here when we put first thing on the pending heap, set the attn bit.

023743   if zero l6.dcb + dcb.priority -> mar(r) ;read the priority (again)
023744   17 -> g1.temp                 ;[1] get a mask
023745   20 -> g2.temp                 ;[2] get a mask

023746   g2.temp & mbr ls              ;check msb of priority for later

023747   g1.temp & mbr -> g1.temp      ;priority only
023750   g1.temp + rb.bits -> base
023751   10 -> temp                    ;get mask to set the bit
023752   rb.ioos -> base               ;fix BASE

023753   if zero 110.attn1 ! temp -> 110.attn1 ;put word back w/ bit set
023754   l2.ret2 -> upc                ;return to caller
023755   ifnot zero 111.attn2 ! temp -> 111.attn2
```

```
        ;IO.DD.CALL, Level 2

        ;Enter here with offset in TEMP. Find the right device driver (DD)
        ;to call. Call it with thw following conventions:
        ; BASE = DCB.RB if it exists, else with R8.MAIN
        ; G6   = DCB address (L6.DCB)
        ; G0   = something to be passed to DD (optional) - i.e., we dont tras

        ;All device drivers return to "DD.RET".

        io.dd.call:
 023756  l6.dcb + dcb.type -> mar(r)     ;find right DD
 023757  nop                            ;[1]
 023760  type.all -> g2.temp             ;[2]

 023761  g2.temp & mbr -> g2.temp, mir ls       ;isolate type bits (low 6)
 023762  if zero l2.ret2 -> upc                 ;if type = 0, forget it
 023763  ifnot zero g2.temp - (dev.max + 1) ls  ;in range?
 023764  ifnot neg baddev -> upc

 023765  if neg disp(iowhat) -> g1.temp         ;get address of table (in low

 023766  l6.dcb + dcb.rb -> mar(r)       ;read the any register block addr
 023767  l6.dcb -> mir                   ;[1] also copy L6 to G6 for DD
 023770  mir -> g6.temp                  ;[2]
 023771  rb.all -> g2.temp               ;(extended constant)

 023772  g2.temp & mbr -> base ls        ;only R8 part wanted
 023773  g1.temp + temp -> upc           ;[1] call device driver's table entry
 023774  (.+1) -> upc                    ;[2] splice, (table is compressed)
                                         ;[3] table entry branches
 023775  if zero rb.main -> base         ;[4] if zero, use MAIN
                                         ;[5] first cycle of driver's routine

        ;return here...

        dd.ret:
 023776  rb.ioos -> base                 ;fix base
 023777  l2.ret2 -> upc                  ;return to caller
 024000  nop
```

```
        ;Subroutine IO.FIND.PHEDR, level 1

        ;Subroutine to compute the address of the right pending queue
        ;header given a priority in G0.TEMP. Puts winning header in G1.TEMP.

        io.find.phedr:
024001   g0.temp -> temp                    ;copy to better place
024002   g0.temp + temp -> g0.temp, temp    ;compute 2*priority
024003   g0.temp + temp -> temp             ;compute 4*priority
024004   10.penh + temp -> temp             ;compute address of header
024005   11.ret1 -> upc                     ;return
024006   temp -> g1.temp                    ;copy answer
```

```
        ;Enter here when we need to scan DCB's to see which to run
        ;next. This occurs whenever we put something new on the pending
        ;queue, or take ourselves off. Enter with BASE pointing at
        ;RB.IOOS.  First, check to process the goad queue, then do a scan.

        pm.scan:
 024007   l12.sgq -> mir ls                ;anything to goad?
 024010   if zero scan.scan -> upc         ;no, just reschedule
 024011   ifnot zero l12.sgq + dcb.goad -> mar(r) ;yes, start fwd ptr fetch
 024012   mir -> l6.dcb                    ;[1] for now
 024013   state.repoked -> g6.temp         ;[2] for SCAN.FROM.PENDING
 024014   g3.rone + mbr -> temp ls         ;this the end?
 024015   0 -> mbr                         ;clear goad pointer
 024016   l6.dcb + dcb.goad -> mar(w)
 024017   if zero 0 -> l12.sgq             ;[1] was end, clear it
 024020   if zero 0 -> l13.egq             ;[2]
 024021   l6.dcb + dcb.state -> mar(r)     ;go look at state word
 024022   ifnot zero temp -> l12.sgq       ;[1] wasn't end, write address+1 into
 024023   ifnot zero l12.sgq - 1 -> l12.sgq ;[2] and fix it
 024024   mbr -> g2.temp
 024025   g2.temp & state.pending ls       ;currently pending?
 024026   ifnot zero scan.from.pending -> upc ;yes, easy
 024027   g2.temp & ~state.skip -> mbr     ;turn off STATE.SKIP bit in memory on
 024030   l6.dcb + dcb.state -> mar(w)
 024031   g2.temp & state.skip ls          ;[1] want to skip?
 024032   if zero scan.noskip -> upc       ;[2] no
 024033   ifnot zero l6.dcb + dcb.pc -> mar(r) ;yes, start fetch of PC
 024034   rb.main -> base                  ;[1] is this current process?
 024035   l6.curr - mir ls                 ;[2]
 024036   g3.rone + mbr -> mbr
 024037   mar -> mar(w)                    ;start write
 024040   if zero l0.pc + 1 -> l0.pc       ;[1] yes, update live copy
 024041   rb.ioos -> base                  ;[2] and fix base

        scan.noskip:
 024042   g2.temp & state.timing ls
 024043   ifnot zero scan.from.timing -> upc ;[1]

        scan.from.ready:
 024044   if zero io.rmq.ready -> upc      ;[2]
 024045   scan.to.pending -> l2.ret2

        scan.from.timing:
 024046   io.rmq.timing -> upc
 024047   scan.to.pending -> l2.ret2

        scan.from.pending:
 024050   g6.temp ! mbr -> mbr             ;turn on STATE.REPOKED bit, clear STA
 024051   scan.bottom -> upc               ;all done
 024052   l6.dcb + dcb.state -> mar(w)

        scan.to.pending:
 024053   io.enq.pend2 -> upc              ;put it on the pending queue, leave r
 024054   scan.bottom -> l2.ret2           ;alone

        scan.bottom:
 024055   ifnot intp pm.scan -> upc        ;[1, from.pending] if not ints, fine
```

```
024056    nop                          ;[2] this is sometimes executed
024057    int.scan -> upc              ;remember about the scan to complete
024060    dummy -> l1.ret1             ;and do this interrupt now, eventuall
```

```
                scan.scan:
024061           rb.main -> base
024062           l10.flags ls                   ;are we INH'ed?
024063           if neg scan.same -> upc        ;if yes, just continue along
024064           ifnot neg rb.ioos -> base      ;else switch bases to IOOS for ATTN m

024065           l10.attn1 -> temp              ;anything in this mask?
024066           temp -> g0.temp ls             ;copy over for JFFO call
024067           if zero scan.attn2 -> upc      ;nope, try next

                scan.attn1:
02407C           ifnot zero jffo -> upc         ;[1] find first one in G0
024071           scan.have.priority -> l1.ret1  ;[1] return to have.priority (in G0)

                scan.attn2:
024072           l11.attn2 -> temp              ;anything in this one?
024073           temp -> g0.temp ls             ;anything in this mask?
024074           if zero crash -> upc           ;if not, nothing at all to run!
024075           if zero ioerr.ntr -> temp      ; ++ nothing to run
024076           jffo -> upc                    ;find priority
024077           (.+1) -> l1.ret1
02410C           g0.temp + 16 -> g0.temp        ;lower priority by one word's worth

                scan.have.priority:            ;(enter here from SCAN.ATTNx)
024101           io.find.phedr -> upc           ;using first one's bit position in G0
024102           (.+1) -> l1.ret1               ;[1] compute the pending header addre

024103           g1.temp + q.forw -> mar(r)     ;look into the first entry on the que
024104           rb.main -> base                ;[1] get at L6.CURR
024105           ioerr.pqe -> temp              ;[2] in case we go to CRASH, below
024106           g1.temp - mbr ls               ;is it the queue header?
024107           if zero crash -> upc

02411C           l6.curr - mbr ls               ;are we switching DCBs?
024111           if zero scan.same -> upc       ;nope, can't possibly need to swap re
024112           ifnot zero mbr -> g1.temp      ;copy over for SCAN.NEW and fall into
```

```
                                        ;from previous page
        ;Fall into here with G1.TEMP having DCB to run. BASE is set to RB.MAI
        ;L6.CURR is 0, it means there is no previous DCB.

        scan.new:                       ;here when we are switching to a new
024113   l6.curr ls                     ;if l6.curr = 0?
024114   if zero scan.new.init -> upc   ;if so, we're init'ing

024115   ifnot zero 10.pc -> mbr        ;write the PC
024116   l6.curr + dcb.pc -> mar(w)     ;make MAR step through context area o
024117   nop
024120   nop                            ;[2]
024121   13.a -> mbr                    ;write the AC
024122   l6.curr + dcb.a -> mar(w)
024123   nop                            ;[1]
024124   nop                            ;[2]
024125   15.x -> mbr                    ;write the index
024126   l6.curr + dcb.x -> mar(w)
024127   nop                            ;[1]
024130   nop                            ;[2]
024131   17.cbit -> mbr                 ;write the flags
024132   l6.curr + dcb.f -> mar(w)
024133   nop                            ;[1]
024134   nop                            ;[2]
024135   14.b -> mbr                    ;write B register
024136   l6.curr + dcb.b -> mar(w)
024137   nop                            ;[1]
024140   nop                            ;[2]
024141   111.sp -> mbr                  ;write SP register
024142   l6.curr + dcb.sp -> mar(w)
024143   nop                            ;[1]
024144   nop                            ;[2]

        ;Now save R1,R2

024145   1 -> mar(r)                    ;read R1
024146   nop                            ;[1]
024147   nop                            ;[2]
024150   mbr -> mbr                     ;propagate EDAC
024151   l6.curr + dcb.r1 -> mar(w)
024152   nop                            ;[1]
024153   nop                            ;[2]
024154   2 -> mar(r)                    ;read R2
024155   nop                            ;[1]
024156   nop                            ;[2]
024157   mbr -> mbr                     ;propagate EDAC
024160   l6.curr + dcb.r2 -> mar(w)

         ;fall into SCAN.NEW.INIT
```

```
                ;from previous page, dangling MAR

                scan.new.init:                  ;(enter here if there was no previous
 024161          g1.temp -> temp                ;[1] copy new DCB to current one
 024162          temp -> 16.curr                ;[2]
 024163          16.curr + dcb.pc -> mar(r)     ;read the PC
 024164          nop                            ;[1]
 024165          nop
 024166          16.curr + dcb.a -> mar(r)
 024167          mbr -> 10.pc                   ;[1] restore PC before MBR changes
 024170          nop                            ;[2]
 024171          mbr -> 13.a                    ;restore it
 024172          16.curr + dcb.x -> mar(r)
 024173          nop                            ;[1]
 024174          nop                            ;[2]
 024175          mbr -> mbr, 15.x
 024176          0 -> mar(w)
 024177          nop                            ;[1]
 024200          nop                            ;[2]
 024201          16.curr + dcb.sp -> mar(r)
 024202          nop                            ;[1]
 024203          nop                            ;[2]
 024204          16.curr + dcb.b -> mar(r)
 024205          mbr -> 111.sp                  ;[1] restore SP before it changes
 024206          nop                            ;[2]
 024207          16.curr + dcb.f -> mar(r)
 024210          mbr -> 14.b                    ;[1] restore B before it changes
 024211          nop                            ;[2]
 024212          16.curr + dcb.r1 -> mar(r)
 024213          mbr -> 17.cbit                 ;[1] restore C-bit before MBR changes
 024214          nop                            ;[2]
 024215          mbr -> mbr                     ;fix EDAC
 024216          1 -> mar(w)
 024217          nop                            ;[1]
 024220          nop                            ;[2]
 024221          16.curr + dcb.r2 -> mar(r)
 024222          nop                            ;[1]
 024223          nop                            ;[2]
 024224          mbr -> mbr                     ;fix EDAC
 024225          2 -> mar(w)
 024226          nop                            ;[1]
 024227          nop                            ;[2]

                scan.same:                      ;(come here from above, too, base RB.
 024230          10.pc -> mar(r)                ;re-fetch the next instruction
 024231          ints -> upc                    ;[1] process it
 024232          nop                            ;[2] [1]
```

```
        ;IO.CLOCK, overlay

        ;Here from USYS at a every 1.6 ms.

        io.clock:
 024233  rb.ioos -> base                ;G7.BASE is buffering application's B
 024234  l15.time + 1 -> l15.time       ;count this tick

        ;Increment L15.TIME and compare it with thing on top of the timing
        ;heap (if anything's there). If L15 .eq. heap, then remove the DCB
        ;and queue it onto its PENDING heap. We do not use GOAD for this
        ;as GOAD will bump the PC and we want to run the process at its
        ;RFI+1, not +2.

 024235  l5.timh -> temp ls
 024236  ifnot zero l5.timh + q.forw -> mar(r)     ;read the timing heap's fo
 024237  if zero clk.wr1 -> upc         ;[1]
 024240  if zero rb.clock -> base       ;[2]

        clock.again:                    ;enter here from below if we need to
 024241  mbr -> l6.dcb                  ;move to useful place
 024242  l6.dcb + dcb.time -> mar(r)    ;go read its timestamp (might be unne
 024243  l6.dcb - temp ls               ;[1] is this empty?
 024244  if zero clock.exit -> upc      ;[2] if so, return

 024245  ifnot zero l15.time - mbr ls   ;compare the two
 024246  if neg clock.exit -> upc       ;if in future, return to USYS

 024247  if zero io.goad -> upc         ;else if just got here, goad the proc
 024250  (.+1) -> l1.ret1

 024251  l6.dcb + q.forw -> mar(r)      ;begin to look at next
 024252  clock.again -> upc             ;[1]
 024253  l5.timh -> temp ls             ;[2] for later

        clock.exit:                     ;out of loop for speed
 024254  clk.wr1 -> upc
 024255  rb.clock -> base

 024256   unext == .

 021325          uram clk.wrret        ;patch into USYS
 021325          io.clock -> upc       ;with this instruction

 024256   uram unext
```

```
024256   unext == .

                                        ;End of IOOS.MIC
        # 20 "m.mic"
```

```
# 1 "io.mic"
```

```
        ;Native-Mode C/30 IO Controller

        ;This file contains code to interface the process manager (PM)
        ;with the various device drivers (DDs).

        ;Subroutines of the form
        ;IODD.xxxx are called from DDs directly. The return address is
        ;passed through in G6.TEMP and BASE is preserved. Since they are
        ;called at interrupt level as well, they must be very careful about
        ;changing variables in the R8.IOOS block, although L1-L3.RET3 are
        ;fair game.

        ;Subroutines of the form IO.xxxx are called from other environments
        ;with BASE pointing to R8.IOOS.

        ;Edit History
        ;
        ; Who    When                    What
        ;
        ; EAH   9-Feb-82        Added IODD.NPUT entry point to IODD.PUT to al
        ;                       device drivers to rid themselves of IOCBs wit
        ;                       causing completion interrupts.  This is so XD
        ;                       to reset devices won't get the process re-pok

        ;Known bugs/deficiencies
```

```
        ;IOCB Format in Main Memory

        ;                           +-----------------+
        ;q.forw                     ! forward pointer! \
        ;                           +-----------------+  \
        ;q.back                     !backward pointer!   ) Defined in QUEUE.MIC
        ;                           +-----------------+  /
        ;q.hedr                     ! header pointer ! /
        ;                           +-----------------+
        ;iocb.size                  !xfer size (bits)!
        ;                           +-----------------+
        ;iocb.addr                  !transfer address!
        ;                           +-----------------+
        ;iocb.flags                 !ae           cccc!
        ;                           +-----------------+

000003  iocb.size     == 3
000004  iocb.addr     == 4
000005  iocb.flags    == 5

        ;In IOCB.FLAGS:

100000  flags.ialways == 100000          ;interrupt always on this IOCB
040000  flags.ierror  == 040000          ;interrupt on non-zero completion cod
000017  flags.ccode   == 000017          ;completion code (0 = normal, non-zer
140017  flags.io      == (flags.ialways!flags.ierror!flags.ccode)

024256   uram unext
```

```
        ;IODD.GET, Subroutine, Level G6

        ;Enter here to try to get a new IOCB in DCB.IOCB. If we get one, we r
        ;the IOCB's address in G0.TEMP. BASE is preserved. We will never supp
        ;one if the associated PCB is IDLE. Enter with DCB addr in TEMP.

        iodd.get:
024256   rb.iocs -> base                ;find current DCB
024257   base -> l16.ddbase             ;save old base
024260   temp -> l6.dcb                 ;set up L6.DCB from caller's DCB addr
024261   l6.dcb + dcb.iocb -> mar(r)    ;look to be sure DCB.IOCB is zero now
024262   g6.temp -> temp                ;[1] save return address
024263   temp -> l7.ddupc               ;[2] in DDUPC, too

024264   mbr ls
024265   ifnot zero io.crash -> upc        ;better be
024266   ifnot zero ioerr.get2 -> temp  ; ++ attempt to get two IOCBs current

024267   l6.dcb + dcb.state -> mar(r)   ;make sure the PCB is not IDLE
024270   state.idle -> g2.temp          ;[1] get a good constant for testing
024271   nop                            ;[2]
024272   g2.temp & mbr ls               ;is it idle?
024273   ifnot zero iodd.ret -> upc     ;yes - return with nothing
024274   ifnot zero 0 ls                ;force ZERO flag in ALUST o

024275   l6.dcb + dcb.get -> mar(r)     ;read the get queue pointer
024276   nop                            ;[1]
024277   nop                            ;[2]
024300   mbr -> g1.temp                 ;into G1.TEMP for Q.DEQ
024301   q.deq -> upc                   ;call DEQ
024302   (.+1) -> l1.ret1

024303   if zero iodd.ret -> upc        ;if got nothing, return

        ;got an IOCB, address in G0.TEMP. Set up DCB.IOCB and clear the compl
        ;code of the IOCB.FLAGS word.

024304   ifnot zero g0.temp + iocb.flags -> mar(r) ;read it first
024305   ~flags.ccode -> g2.temp        ;[1] get a constant for clearing it
024306   nop                            ;[2]
024307   g2.temp & mbr -> mbr           ;clear the bits
024310   g0.temp + iocb.flags -> mar(w) ;write it back
024311   nop                            ;[1]
024312   nop                            ;[2]

024313   g0.temp -> mbr                 ;write IOCB address
024314   l6.dcb + dcb.iocb -> mar(w) ls ;into DCB.IOCB and clear the ZERO con

        ;Here to return to a device driver whose BASE has been cached in L16.
        ;and whose return UPC is in L7.DDUPC. Does not alter ALUST because ZE
        ;flag is often returned to caller.

        iodd.ret:
024315   l7.ddupc -> upc                ;return
024316   l16.ddbase -> base             ;and return base
```

```
        ;IODD.PUT, Subroutine, Level G6

        ;Subroutine to put the current IOCB (DCB.IOCB) on the put list and ch
        ;for possible interrupt generation on the macroprocess.
        ;is used as a flag throughout this code to determine whether IO.GOAD
        ;be called at its end. Enter at IODD.IPUT to start off assuming a
        ;macrointerrupt is required. Caller places DCB in TEMP.

        iodd.nput:
 024317  rb.ioos -> base
 024320  base -> 116.ddbase
 024321  iodd.put2 -> upc
 024322  1 -> 13.ret3                    ;a 1 in L3 means never interrupt

        iodd.iput:
 024323  rb.ioos -> base
 024324  base -> 116.ddbase
 024325  iodd.put2 -> upc
 024326  -1 -> 13.ret3                   ;a -1 in L3 means always interrupt

        iodd.put:
 024327  rb.ioos -> base                 ;switch bases
 024330  base -> 116.ddbase              ;save old one
 024331  0 -> 13.ret3                    ;a 0 in L3 means figure it out here

        iodd.put2:                      ;(enter here from IODD.IPUT)
 024332  temp -> 16.dcb                 ;set up DCB from caller's arg in TEMP
 024333  16.dcb + dcb.iocb -> mar(r)     ;get pointer to current IOCB
 024334  g6.temp -> temp                 ;[1] save caller's UPC
 024335  temp -> 17.ddupc                ;[2]
 024336  mbr -> g0.temp ls               ;move it into G0.TEMP for Q.ENQ and c

 024337  0 -> mbr                        ;then clear pointer to it
 024340  16.dcb + dcb.iocb -> mar(w)
 024341  if zero io.crash -> upc           ;[1] if zero, there wasn't one!
 024342  if zero ioerr.put0 -> temp      ;[2]  ++ attempt to put a zero IOCB

 024343  16.dcb + dcb.put -> mar(r)      ;get pointer to put queue
 024344  iodd.put3 -> 11.ret1            ;[1] set up for returning from Q.ENQ
 024345  q.enq -> upc                    ;[2] call it
 024346  mbr -> g1.temp                  ;set up arg for ENQ

        iodd.put3:                      ;here after ENQ
 024347  13.ret3 ls                      ;is there a need to interrupt already
 024350  if neg iodd.put.int -> upc      ;yes, interrupt
 024351  if zero g0.temp + iocb.flags -> mar(r) ;no, if L3 = 0, check IOCB
 024352  ifnot zero iodd.ret -> upc      ;[1] if L3 = 1, just return
 024353  (flags.ialways ! flags.ierror) -> g2.temp ;[2]

 024354  g2.temp & mbr ls                ;interrupt always or error?
 024355  if neg iodd.put.int -> upc      ;if sign=1, yes
 024356  if zero iodd.ret -> upc         ;if zero, return now
 024357  mbr -> g2.temp
 024360  ifnot zero g2.temp & flags.ccode ls ;non-zero condition code?
 024361  if zero iodd.ret -> upc         ;if zero, return now
 024362  nop                             ;[1]
```

```
        ;Here to cause an interrupt.

        iodd.put.int:
024363   io.goadskip -> upc             ;goad this process
024364   iodd.ret -> l1.ret1            ;return when done

        iodd.goadskip:                  ; here from outside world to cause
024365   rb.iocs -> base                ; an interrupt without hacking iocb's
024366   base -> l16.ddbase
024367   iodd.put.int -> upc
02437C   temp -> l6.dcb

        io.crash:
024371   crash -> upc                   ;we do this so the iodevice's regs
024372   l16.ddbase -> base             ;will get saved in crash.xxx
```

```
        ;IO.CLEAR, Subroutine, Level 2

        ;Enter with BASE=RB.IOOS.
        ;Subroutine to master clear all device drivers. Goes through each dev
        ;in RB.DEV (until DEV.MAX is reached), calling it at it's DD.CLR.OFFS
        ;Note that we can't go through the IO.DD.CALL logic because there's n
        ;current DCB, etc. Also, the DD will return directly to DD.CLR.RET.
        ;The DD is called with RB.TEMP.


        io.clear:

 024373  m2.icreset -> misc2              ;clear the IO world
 024374  nop                             ;how much time should we wait?
 024375  l2.ret2 -> upc                  ;this dangles into first of following


        ;Here from anywhere in any device driver to return from an interrupt
        ;in the standard way.  One cycle is available before INTS is read.
        ;Two cycles are available before mbr is changed.


 024405  fill 7,1                        ; seven unorthodox no-ops.

        io.ret:

 024406  g7.base -> base                 ;<M> [N-1] restore base
 024407  ints -> upc                     ;[N] return
 024410  g13.mbr -> mbr                  ;restore MBR
```

```
        ;Enter here if 10.dcb is valid for this device.  Otherwise, same as i
        ;Don't come here if device might be xdv'd when not enabled.  MIR clob

        iodd.xdv:

024411   10.dcb -> mir                      ;check register blocks
024412   g6.temp - mir ls
024413   ifnot zero crash -> upc
024414   ifnot zero ioerr.rbmu -> temp  ; ++ Register Block Messed Up ++


        ;The standard xdv dispatch.  Come here with address of table in TEMP,
        ;number of entries (max index + 1) in MAR, and index in G0.TEMP, as p
        ;by io316.xdv.  Range is checked, proper point in driver is entered,
        ;g6.temp recieves a copy of device register base.

        io.xdv:

        ;Range check the XDV code (passed though in G0.TEMP) and dispatch.

024415   g0.temp ls                         ;make sure we like code
024416   if neg io.bxa1 -> upc

024417   ifnot neg g0.temp - mar ls
024420   if neg g0.temp + temp -> upc    ;[1] begin to dispatch
024421   if neg (.+1) -> upc             ;[2] ([3] in table)
024422   if neg base -> g6.temp          ;[4] save BASE for manipulations in d

        io.bxa1:
024423   crash -> upc
024424   ioerr.bxa1 -> temp              ; ++ Bad XDV arg #1 (AC) ++

024425   unext == .
        # 21 "m.mic"
```

```
# 1 "cac.mic"
```

```
; Console And Cassette Port Device Driver for the NMFS C/30 IMP I/O S
;
;                        Glenn M. Simpson
;                         March 24, 1982
;
;
; In this file the following prefixes appear often.
;
;        CAC- Console And Cassette routines
;       ICAC- Input side routines
;       OCAC- Output side routines
;        CON- Console specific
;        CAS- Cassette specific
;
;
; MAJOR COMPONENTS:
;
;       1) Decoding and emulation of device-specific
;          instructions.
;       2) Hardware and software mappings, and definitions.
;       3) Service routines that interact with USYS
;          device handlers to receive and transmit data.
;
;
; I/O INSTRUCTIONS:  each is prefixed with CAC, eg. cac.apr.
;                    BASE=device driver's register set pointer,
;                    G6.TEMP=DCB pointer, G0.TEMP=Device command,
;                    B-register=data, A=-1-error, 0-done, NC-busy.
;
;       APR (activate process): Bind process--identified by DCB
;                               pointer--to a hardware device.
;                               Initialize software environment.
;
;       DPR (delete process):   Unbind process from device.
;                               Terminate current I/O.
;
;       XDV (execute device):   Execute various device functions.
;           X will contain the PCB pointer.  B-register is used to
;           pass operands.  A (accumulator) will contain one of the
;           following command codes (a response, when appropriate,
;           will be returned in A):
;
;           0 - NOP:    No-op.
;
;           1 - RESET:  Relinquish control of port and
;                       terminate current I/O. Both input
;                       and output sides are serviced.
;
;           2 - ENABLE: Take control of port.  Terminate
;                       current I/O then initiate new I/O.
;                       Both input and output sides are serviced.
;
;                       B-reg                OPTION
;
;                       b0=1(0) DDT Enable (Transparent mode).
;                       b1=1(0) Automatic Flow Control (Off).
```

```
;                       b2=1(0) Hi-Lo (Lo-Hi) byte sequence.
;
;           3 - BREAK:  Cause device to transmit <break> signal.
;                       Doesn't preempt a transmission but normal
;                       operation will be suspended subsequently.
;                       No-op when issued by an input process.
;
;           4 - MARK:   Cause device to stop transmitting <break>.
;                       Device resumes normal operation by
;                       holding its output line in <mark> state.
```

```
;           5 - BAUD:   Set baud rate:
;
;                              B-reg      Baud
;                              (oct)      Rate
;                                0        110
;                                1        300
;                                2        1200
;                                3        1800
;                                4        2400
;                                5        4800
;                                6        9600
;                                7        19200
;
;           6 - XFER:   Do single character transfer through B.
;                       Input: return a character in B.
;                       Output: xmit character found in B.
;                       Response: A=-1-error, 0-done, NC-busy.
;
;           7 - ABORT:  Terminate current I/O and put IOCB on the
;                       PUT queue.  I/O will continue with the
;                       next IOCB, if one exists.
;
;          10 - STATUS: Read hardware/software status and control.
;                       Note, 1_N means 1 shifted left N octal bits,
;                       therefore, N identifies the Nth bit of word.
;
;                           tty.trdy == 1_0     2651 transmitter ready.
;                           tty.rrdy == 1_1      "   receiver ready.
;                           tty.temt == 1_2      "   transmitter empty.
;                           tty.prty == 1_3      "   input parity error.
;                           tty.ovrn == 1_4      "   input overrun error.
;                           tty.brk  == 1_5      "   input framing error.
;                           tty.dcd  == 1_6      "   data carrier detected.
;                           tty.dsr  == 1_7      "   data set ready.
;
;                         cac.csr.xfr == 1_10   Software "transfer" flag-
;                                                Input: non-IOCB data receive
;                                               Output: transmission in progr
;                         cac.csr.ovr == 1_11   Input overrun--transfer occur
;                                               while CAC.CSR.XFR was still o
;                                               (Available from input side on
;
;          11 - CLEAR ERR:  Reset hardware error flags and software
;                           status flags.
;
;        POV (poke device)  Start I/O by fetching an IOCB.
;                           A no-op if no IOCB.
;
```

```
                ; Configuration:

000005          caci == 5                       ; Input side device type.
000006          caco == 6                       ; Output side device type.

                ; Completion codes: returned in IOCB completion code field.

000001          cac.cc.abt == 1                 ; IOCB aborted (ABORT, RESET, DPR).
000002          cac.cc.ovr == 2                 ; Device input overrun.
000003          cac.cc.brk == 3                 ; Device received <break>.

                ; Register block allocation:

000400          rb.cac == 400                   ; Base for device driver routines.
                                                ; 16 words/block, one block per side.
000400          rb.icon == rb.cac + 0           ; Console input side allocation,
000420          rb.ocon == rb.cac + 20          ;         output side allocation.
                                                ; Cassette allocation:
000440          rb.icas == rb.cac + 40          ;         input side,
000460          rb.ocas == rb.cac + 60          ;         output side.

000020          caci2o == rb.ocon - rb.icon     ; Displacement parameter for
                                                ;  switching between environments.

                ; Local register definitions: (Output side L16 and L17 are not used)

000000          l0.dcb  == 10                   ; Associated DCB.
000001          l1.ret1 == 11                   ; Level 1 subroutine return link.
000002          l2.ret2 == 12                   ; Level 2 subroutine return link.
000003          l3.iocb == 13                   ; IOCB (0=none).
000004          l4.addr == 14                   ; Macro-memory address for next trans
000005          l5.left == 15                   ; Number of remaining transfers.
000006          l6.char == 16                   ; Local character buffer.
000007          l7.next == 17                   ; UPC of service routine for next byt
000010          l10.csr == 110                  ; Device driver control and status wo
000011          l11.tmp == 111                  ; Useful temporary register.
000012          l12.dev == 112                  ; Base for associated USYS device han
000013          l13.inp == 113                  ; Input side register block base.
000014          l14.svc == 114                  ; UPC of Lo-byte/hi-byte service rout
000015          l15.bgn == 115                  ; Starting UPC of state machine.
000016           l16.stp == 116                 ; (i) ^S (Control-S) ascii code.
000016           l16.buf == 116                 ; (o) Buffer for single char transfer
000017          l17.ctn == 117                  ; (i) ^Q (Control-Q) ascii code.
```

```
        ; Device driver status (in LO.CSR):  (i)-input, (o)-output, (b)-both.

000400   cac.csr.xfr == 1_10            ; (i) Input received a character when
                                        ;     when block I/O was not in progr
                                        ; (o) Output is transmitting a charac

001000   cac.csr.ovr == 1_11            ; (i) Input overrun.

002000   cac.csr.err == 1_12            ; (i) Hardware malfunctioned.

004000   cac.csr.afc == 1_13            ; (i) Automatic flow control is on.
                                        ; (o) Discontinue transfers.

010000   cac.csr.ena == 1_14            ; (b) Device driver has been enabled.

020000   cac.csr.out == 1_15            ; (o) Buffer for single character
                                        ;     transfers is full.

034400   cac.csr.ctl == (cac.csr.xfr!cac.csr.afc!cac.csr.ena!cac.csr.out)
037400   cac.csr.all == (cac.csr.ctl!cac.csr.ovr!cac.csr.err)


        ; Device options: passed in B-REG on an XDV.ENABLE.  Enabling
        ; DDT allows interpretation of ^N and E to switch control of
        ; terminal between uDDT and the running process.  Automatic
        ; Flow Control allows the interpretation of ^S and ^Q to
        ; halt and continue transmitting characters to the console
        ; from its device driver.  The HI-LO bit determines the sequence
        ; in which bytes will be serviced.

000001   cac.opt.ddt == 1_0             ; Enable DDT.
000002   cac.opt.afc == 1_1             ; Auto Flow Control.
000004   cac.opt.hib == 1_2             ; Service hi-byte first.


        ; 2651 status register: error flags in USYS L5.IN.STAT.

000020   dev.sts.ovr == 1_4             ; Overrun error--input transfer occur
                                        ;  while receive holding register had
000040   dev.sts.brk == 1_5             ; Receiver detected <break>.

000060   dev.sts.err == (dev.sts.ovr!dev.sts.brk); Device errors.


        ; 2651 control register:

000010   dev.cmd.brk == 1_3             ; Force <break>.


        ; 2651 mode register 1: select baud rate and specify
        ;                       internal xmt/rcv clocks.

000062   baud.110 == 62
000065   baud.300 == 65
000067   baud.1200 == 67
000070   baud.1800 == 70
000072   baud.2400 == 72
```

```
000074    baud.4800 == 74
000076    baud.9600 == 76
000077    baud.19200 == 77


          ; Error codes: used by CRASH utility to identify nature
          ;              of malfunction.

000100    cacerr.0 == 100                   ; Base for CAC errors.
000100    cacerr.nena == cacerr.0 + 0       ; Device driver not enabled.
000101    cacerr.ibrr == cacerr.0 + 1       ; Illegal baud rate request.
```

```
        ; Overlays: vectors to input and output service routines.
        ;           Overwrites corresponding code area in USYS (uram).

 021356 uram ttyrecdum
 021356  icac.int -> upc                 ; Go to input side interrupt handler.
 021357   mbr -> g13.mbr                 ; Save MBR to make code reentrant.

 021354 uram ttyxdondum
 021354  ocac.int -> upc                 ; Go to output side interrupt handler
 021355   mbr -> g13.mbr                 ; As usual, save MBR.


        ; Function dispatch:

 024425 uram unext

        icac.0:                         ; Input side dispatch.

 024425  cac.apri -> upc
 024426  cac.dpr -> upc
 024427  cac.xdvi -> upc
 024430  cac.pdv -> upc

        ocac.0:                         ; Output side dispatch.

 024431  cac.apro -> upc
 024432  cac.dpr -> upc
 024433  cac.xdvo -> upc
 024434  cac.pdv -> upc
```

```
        ; INPUT INTERRUPT HANDLER.
        ;
        ; If device driver has been turned on by an XDV.ENABLE, the input
        ; service routine vector will cause transfer to this entry point
        ; whenever USYS receives a character.  The character will be
        ; passed in TEMP.

        icac.int:                           ; Here to service byte from USYS.
 024435  L15.ddt - rb.cpdt ls               ; Who poked us?
 024436  if zero rb.icon -> base             ; Fix base accordingly.
 024437  ifnot zero rb.icas -> base


        ; Flow control operations.

 024440  110.csr & cac.csr.afc ls             ; Flow control on?
 024441  if zero icac.nfc -> upc             ; No flow control, proceed as usual.
 024442  ifnot zero 116.stp - temp ls        ; Else, check for Control-S character
 024443  if zero 113.inp + caci2o -> base; Switch to output side.
 024444  if zero cac.ret -> upc             ; Dismiss and
 024445   if zero 110.csr ! cac.csr.afc -> 110.csr;  halt output transfers.

 024446  117.ctn - temp ls                  ; Is it a Control-Q character?
 024447  ifnot zero icac.nfc -> upc         ; No control chars, proceed as usual.
 024450  if zero 113.inp + caci2o -> base; Else, switch to output side.
 024451  110.csr & ~cac.csr.afc -> 110.csr; Clear flag to continue output.
 024452  110.csr & cac.csr.xfr ls            ; Is there an output transfer in pro
 024453  ifnot zero cac.ret -> upc          ; If there is, dismiss now.
 024454  if zero 13.iocb ls                 ; Check for an active IOCB.
 024455  if zero cac.ret -> upc             ; If none, just dismiss.
 024456  ifnot zero 17.next -> upc          ; Otherwise, go service the next outp
 024457   cac.ret -> 11.ret1                ;  and eventually dismiss.


        ; Normal sequence when no flow control processing is required.

        icac.nfc:                           ; Here when no flow control.
 024460  13.iocb ls                         ; Is there an active IOCB?
 024461  ifnot zero 17.next -> upc          ; An IOCB is available.  Go service b
 024462   ifnot zero 15.left - 1C -> 15.left; Update transfer count.

 024463  110.csr & cac.csr.xfr ls            ; Else, check for character overrun.
 024464  ifnot zero 110.csr ! cac.csr.ovr -> 110.csr; Didn't read previous ch
 024465  if zero 110.csr ! cac.csr.xfr -> 110.csr; No overrun, so set the
                                            ;  "transfer" (rcvd input) flag.
 024466  cac.ret -> upc                     ; Dismiss and
 024467   temp -> 16.char                   ;  save character.


        ; Come here when new input I/O is initiated by an XDV.ENABLE or
        ; a PDV.  If a character was received prior to IOCB acquisition,
        ; the saved character will be immediately stored in the IOCB
        ; buffer.  Multiple transfers will result in an overflow error
        ; condition whenever an IOCB is not yet available.  The overflow
        ; error, or any detected hardware error, will cause the new IOCB
        ; to be discarded with the appropriate completion code.
```

```
                icac.bgn:                       ; Here to start I/O with new IOCB.
024470           if zero dd.ret -> upc          ; No single-mode transfers have occur
024471            if zero icac.1st -> 17.next   ;  so return expecting the first byte

024472           110.csr & cac.csr.ovr ls       ; Did overflow occur?
024473           ifnot zero icac.nok -> upc     ; Yes, must discard IOCB.
024474            ifnot zero cac.cc.ovr -> g0.temp; Provide "overrun" completion code

024475           15.left - 10 -> 15.left        ; No overflow, so just update transfe
                                                ;  and proceed with servicing transfe
024476           icac.tst -> upc                ; Test device for errors.
024477            (.+1) -> temp
024500           if zero dd.ret -> upc          ; No errors, so return to caller and
024501            if zero icac.2nd -> 17.next   ;  service the second byte next time.

                icac.nok:                       ; Error occurred before I/O began.
024502           110.csr ! cac.csr.err -> 110.csr; Set flag for device failure.
024503           0 -> temp                      ; Pad a "0" in place of second byte.
024504           114.svc -> upc                 ; Go store word.
024505            dd.ret -> 12.ret2             ; Return to caller when done.


                ; Entry point for servicing the first transfer.  The received
                ; character is passed in TEMP from USYS.

                icac.1st:                       ; Here to service first byte.
024506           temp -> 16.char                ; Save byte.
024507           icac.tst -> upc                ; Go test for device errors.
024510            (.+1) -> temp
024511           ifnot zero icac.bad -> upc     ; Error occurred, so go service bad
024512            ifnot zero 0 -> temp          ;  IOCB and provide padding.

024513           15.left ls                     ; If no malfunctions, is IOCB used up
024514           ifnot zero cac.ret -> upc      ; If more transfers are left, dismiss
024515            ifnot zero icac.2nd -> 17.next;  and do second byte next time.
024516           114.svc -> upc                 ; Else, go service last transfer and
024517            0 -> temp                     ;  pad the second byte position.


                ; Entry point for servicing the second input transfer.  USYS passed
                ; the character in TEMP.

                icac.2nd:                       ; Here to service the second byte.
024520           temp -> 111.tmp                ; Put second byte in safe place.
024521           icac.tst -> upc                ; Go check for device errors.
024522            (.+1) -> temp
024523           ifnot zero icac.bad -> upc     ; Device failed, go service bad IOCB.
024524           111.tmp -> temp                ; Get back the saved byte.
024525           114.svc -> upc                 ; Didn't fail, go service second byte
024526            icac.1st -> 17.next           ; Expect the first byte next time.


                icac.bad:                       ; Error occurred during input I/O.
024527           110.csr ! cac.csr.err -> 110.csr; Set error flag.
024530           114.svc -> upc                 ; Go service erroneous transfer.
024531            cac.ret -> 12.ret2
```

```
                icac.lob:                       ; Here to service lo-byte.
024532           l6.char -> mbr                 ; Copy the saved hi-byte.
024533           smbr -> mbr                    ; Place it in MBR's high-order positi
024534           icac.store -> upc              ; Go store word.
024535            temp -> l6.char               ; Put lo-byte in safe place.

                icac.hib:                       ; Here to service hi-byte.
024536           temp -> mbr                    ; Get the hi-byte.
024537           icac.store -> upc              ; Go store word with hi-byte placed
02454C            smbr -> mbr                   ;  in MBR's high-order position.

        ; Utility routine for detecting device malfuntions.  Device is
        ; checked when new input I/O commences and after each input
        ; transfer.  The error code is returned in G0.TEMP; this code
        ; is subsequently written in the IOCB as the completion code
        ; when a device failure has been detected.

                icac.tst:                       ; Here to check device for errors.
024541           112.dev -> base                ; Switch to USYS to check for errors.
024542           base -> L0.base                ; Save old base.
024543           temp -> L1.ret1                ; Put return link in safe place.
024544           L5.in.stat & dev.sts.err -> temp ls; Did overrun or break occur?
024545           if zero icactst.ext -> upc     ; If no errors, just return.
024546           ifnot zero temp -> g0.temp     ; Otherwise, copy error flags.
024547           g0.temp & dev.sts.ovr ls       ; Was it an overrun error?
02455C           ifnot zero cac.cc.ovr -> g0.temp ls; Yes, provide its completion cod
024551           if zero cac.cc.brk -> g0.temp ls; Break occurred, write correspondin

                icactst.ext:                    ; Here to return from test sequence.
024552           L1.ret1 -> upc                 ; Go write completion code into IOCB.
024553            L0.base -> base               ; First restore base.

        ; Utility routine for storing formed words (lo-byte + hi-byte) into
        ; the macro-memory buffer specified in the IOCB.  L6.CHAR and the MBR
        ; hold the bytes which are combined to form the word which is to be
        ; stored.  In the event that input I/O might complete or terminate
        ; on a byte boundary, the incompleted word will be padded with a
        ; 0 (zero) byte.

                icac.store:                     ; Here to put word into macro-memory.
024554           l6.char ! mbr -> mbr           ; Form word.
024555           l4.addr -> mar(w)              ; Store it in macro-memory buffer.
024556           l4.addr + 1 -> l4.addr         ; [1] Update buffer address.

024557           110.csr & cac.csr.err ls       ; [2] Did device malfunction?
024560           110.csr & ~cac.csr.err -> 110.csr; First clear the flag.
024561           ifnot zero icac.err -> upc     ; If error was flagged, discard IOCB.
024562           if zero l5.left ls             ; Otherwise check for more transfers.
024563           ifnot zero cac.ret -> upc      ; More transfers left, so dismiss.

024564           if zero cac.start -> g6.temp   ; IOCB exhausted, try to get another
024565           cac.done -> upc                ;  after the current one is put away.
024566            (.+1) -> l2.ret2
024567           ifnot zero icac.1st -> l7.next ; Got another IOCB, expect first byte
02457C           cac.ret -> upc                 ; Dismiss.
```

```
024571    if zero icac.bgn -> 17.next   ; If no more IOCBs, reinitialize inpu
                                        ;  state machine.

          icac.err:                     ; A device error was flagged.
024572    cac.scc -> upc                ; Go set appropriate CC and discard I
024573     cac.start -> g6.temp         ; Try to get another IOCB when done.


          ; Interrupt handler for output service requests.

          ocac.int:
024574    L13.ttyflgs & ~tty.own.pk -> L13.ttyflgs; We've been "poked" so clea
024575    L13.ttyflgs & tty.own.m -> temp; Mask off ownership flag.
024576    L15.ddt - rb.cpdt ls          ; Who poked us?
024577    if zero rb.ocon -> base       ; It was either the console or the
024600    ifnot zero rb.ocas -> base    ;  cassette handler in USYS.

024601    l10.csr & ~cac.csr.xfr -> l10.csr; Always clear "transfer" flag.
024602    l10.csr & cac.csr.afc ls       ; Is flow control on?
024603    ifnot zero cac.ret -> upc     ; If so, dismiss now.
024604    if zero l10.csr & cac.csr.out ls; Else, check for pending single tra
024605    ifnot zero ocac.sgl -> upc    ; One is pending, go service it and m
024606     ifnot zero l10.csr & ~cac.csr.out -> l10.csr; certaing flag is rese

024607    l3.iocb ls                    ; Is there an IOCB?
024610    if zero cac.ret -> upc        ; No, dismiss.
024611    ifnot zero temp ls            ; Else, is device ours?
024612    if zero cacxdv.dis -> upc     ; No, go discard current IOCB.
024613     if zero cac.ret -> temp      ; Dismiss when done.

024614    17.next -> upc                ; Yes, go service the next byte
024615     cac.ret -> l1.ret1           ;  then dismiss when done.


          ocac.bgn:                     ; Here to begin I/O with new IOCB.
024616    if zero l10.csr & cac.csr.afc ls; If a transfer is in progress or
024617    ifnot zero dd.ret -> upc      ;  flow control is on, return now.
024620    if zero l12.dev -> base       ; Switch to USYS.
024621    base -> L0.base               ; Save our environment.
024622    L13.ttyflgs & tty.own.m ls    ; Is device ours?
024623    if zero dd.ret -> upc         ; If not, return now.
024624    ifnot zero L0.base -> base    ; Else, fix base then proceed with
024625    dd.ret -> l1.ret1             ;  I/O and return when done.


          ocac.1st:                     ; Here to service the first byte.
024626     ocac.2nd -> 17.next          ; Do the second byte next time.
024627    14.addr -> mar(r)             ; Get a word from buffer.
024630    l14.svc -> upc                ; [1] Go transmit the first byte.
024631     14.addr + 1 -> 14.addr       ; [2] Update macro-memory address.


          ocac.lob:                     ; Here to service the lo-byte.
024632     smbr -> l6.char              ; Save hi-byte for next transfer.
024633     ocac.send -> upc             ; Go transmit lo-byte.
024634      mbr -> temp                 ; Provide the byte.

          ocac.hib:                     ; Here to service the hi-byte.
```

```
024635    mbr -> 16.char                    ; Save lo-byte for next transfer.
024636    ocac.send -> upc                  ; Go transmit hi-byte.
024637     smbr -> temp                     ; Reposition hi-byte in MBR.


          ocac.2nd:                         ; Here to service the second byte.
024640    ocac.1st -> 17.next               ; Do the first byte next time.
024641    16.char -> temp                   ; The second byte will now be
                                            ;  transmitted.

          ocac.send:                        ; Here to transmit a byte.
024642    112.dev -> base                   ; Set up base for driver in USYS.
024643    base -> L0.base                   ; Save our base.

024644    ttyxmit -> upc                    ; Give character to USYS.
024645     (.+1) -> L3.ret3

024646    L13.ttyflgs & tty.out ls          ; Did transfer complete?
024647    L0.base -> base                   ; Restore base first.
02465C    ifnot zero 110.csr ! cac.csr.xfr -> 110.csr; No, so set "transfer" f

024651    15.left - 10 -> 15.left ls        ; Was this the last transmission?
024652    11.ret1 -> upc                    ; Return and eventually transfer to
024653     if zero ocac.end -> 17.next      ;  completion service if last transfe


          ocac.sgl:                         ; Here for pending single char xfrs.
024654    116.buf -> temp                   ; Retrieve saved character.
024655    112.dev -> base                   ; Fix base for USYS device handler.
024656    base -> L0.base                   ; Save pointer to our environment.

024657    ttyxmit -> upc                    ; Give character to USYS.
02466C     (.+1) -> L3.ret3

024661    L13.ttyflgs & tty.out ls          ; Did transfer complete?
024662    L0.base -> base                   ; First restore our environment.
024663    cac.ret -> upc                    ; Dismiss and set "transfer" flag if
024664     ifnot zero 110.csr ! cac.csr.xfr -> 110.csr;  transfer didn't compl


          ocac.end:                         ; Here to begin completion service.
024665    cac.start -> g6.temp              ; Attempt more I/O when done with thi
024666    cac.done -> upc                   ; Go put away used IOCB.
024667     (.+1) -> 12.ret2
02467C    ifnot zero ocac.1st -> upc        ; If we got another IOCB go start out
024671    if zero ocac.bgn -> 17.next       ; Otherwise, will start from beginnin


          cac.ret:                          ; Here to dismiss.
024672    g7.base -> base                   ; First fix base for MAIN then
024673    ints -> upc                       ;  dismiss with MBR restored.
024674     g13.mbr -> mbr


        ; CAC.APR, Microcall: activate process.
        ; Entry point for APRs.  Enter with G6.TEMP pointing to DCB
        ; and BASE=DCB.R8.  This function sets up the DCB back-pointer to eff
```

```
        ; the binding of a process to an I/O device.  Thus creating an I/O
        ; process.

        cac.apri:
 024675  base -> 113.inp
 024676  g6.temp -> temp                 ; Get DCB back-pointer.
 024677  temp -> 10.dcb                  ; Keep a permanent copy.

 024700  0 -> 13.iocb                    ; No IOCB.
 024701  0 -> 110.csr                    ; Clear control and status register.

 024702  113.inp - rb.icon ls            ; Either console or cassette process.
 024703  if zero rb.ctty -> 112.dev      ; Copy pointer to associated USYS dev
 024704  ifnot zero rb.ltty -> 112.dev   ;  handler environment.

 024705  icac.bgn -> 115.bgn             ; UPC for starting buffered input I/O
 024706  ascii..s -> 116.stp             ; ^S (Control-S) ascii code.
 024707  ascii..q -> 117.ctn             ; ^Q (Control-Q) ascii code.
 024710  dd.ret -> upc
 024711   icac.bgn -> 17.next            ; Initialize input state machine.


        cac.apro:
 024712  base -> 113.inp
 024713  g6.temp -> temp                 ; Get DCB back-pointer.
 024714  temp -> 10.dcb                  ; Keep a permanent copy.

 024715  0 -> 13.iocb                    ; No output IOCB.
 024716  0 -> 110.csr                    ; Clear output status.

 024717  rb.ocon -> temp
 024720  113.inp - temp ls               ; Either console or cassette process.
 024721  if zero rb.ctty -> 112.dev      ; Copy pointer to associated USYS dev
 024722  ifnot zero rb.ltty -> 112.dev   ;  handler environment.

 024723  113.inp - caci2o -> 113.inp     ; Keep copy of input side register ba
 024724  dd.ret -> upc
 024725   ocac.bgn -> 115.bgn            ; UPC when new output block I/O is st



        ; CAC.DPR, Microcall: delete process.
        ; Entry point for DPRs.  Enter as usual: G6.TEMP=DCB,
        ; and BASE=DCB.RB.  This function clears the DCB back-pointer to
        ; unbind (kill) the corresponding I/O process.

        cac.dpr:
 024726  10.dcb -> temp                 ; Set up test of DCB back-pointer.
 024727  g6.temp - temp ls               ; Is it the correct pointer?
 024730  ifnot zero crash -> upc         ; If it's not, issue error msg.
 024731   ifnot zero ioerr.rbmu -> temp ; ++ Register Block Messed Up ++

 024732  13.iocb ls                      ; Is there I/O in progress?
 024733  if zero cacdpr.ubd -> upc       ; If not, just unbind I/O process.
 024734  ifnot zero cac.cc.abt -> g0.temp; Else, set up mask for "error" flag
 024735  cac.scc -> upc                  ; Go discard IOCB with flag set.
 024736   (.+1) -> g6.temp
```

```
                cacdpr.ubd:
024737           dd.ret -> upc                  ; Return to caller with
024740            0 -> 10.dcb                   ;  the DCB back-pointer cleared.



                ; CAC.XDV, Microcall: execute device function.
                ; Entry point for consette XDVs.  Enter with G6.TEMP=DCB,
                ; base points to the device driver's register block, and G0.TEMP
                ; holds the command code.  Data is passed in IREG.

                cac.xdvi:
024741            cac.xdv -> upc
024742            0 -> 111.tmp                  ; Flag is reset for input side.

                cac.xdvo:
024743            1 -> 111.tmp                  ; Flag is set for output side.

                cac.xdv:
024744           cacxdv.tbl -> temp
024745           iodd.xdv -> upc
024746           cactbl.siz -> mar

                cacxdv.tbl:

024747           dd.ret -> upc                  ; (0) NOP
024750           cacxdv.rst -> upc              ; (1) RESET
024751           cacxdv.ena -> upc              ; (2) ENABLE
024752           cacxdv.brk -> upc              ; (3) BREAK
024753           cacxdv.mrk -> upc              ; (4) MARK
024754           cacxdv.bps -> upc              ; (5) BAUD
024755           cacxdv.xfr -> upc              ; (6) TRANSFER
024756           cacxdv.abt -> upc              ; (7) ABORT
024757           cacxdv.sts -> upc              ;(10) STATUS
024760           cacxdv.cle -> upc              ;(11) CLEAR ERRORS

000012          cactbl.siz == (. - cacxdv.tbl ) ; Table length calculation.



                ; Device specific functions.

                cacxdv.rst:                    ; Here to relinquish control of devic
024761           113.inp -> temp
024762           110.csr & ~cac.csr.all -> 110.csr; Clear software flags.
024763           112.dev -> base               ; Fix base for USYS.
024764           temp -> L0.base               ; Save our base for input side.
024765           cacxdv.off -> upc             ; Go turn off device.
024766            (.+1) -> temp
024767           L13.ttyflgs & ~(tty.own.m!tty.trans) -> L13.ttyflgs; Give device to
024770           cacxdv.on -> upc              ; Go restore Command Register and cle
024771            (.+1) -> temp                ;  error flags.
024772           L0.base -> base               ; Fix base for input side.

024773           13.iocb ls                    ; Is there an active IOCB?
024774           ifnot zero cacxdv.dis -> upc  ; Yes, go discard it.
024775            ifnot zero (.+1) -> temp
```

```
024776   l13.inp + caci2o -> base        ; Otherwise, switch to output side.
024777   l10.csr & ~cac.csr.all -> l10.csr; Clear flags.
025000   l3.iocb ls                      ; Is there an IOCB here?
025001   if zero dd.ret -> upc           ; No, return now.
025002   ifnot zero dd.ret -> temp       ; Else, will return after IOCB
                                         ;  has been put away.
         cacxdv.dis:                     ; Here to go discard IOCB.
025003    cac.cc.abt -> g0.temp          ; Get "abort" flag.
025004    cac.scc -> upc                 ; Go set flag in IOCB which is then
025005     temp -> g6.temp               ;  discarded.

         cacxdv.off:                     ; Here to turn off device.
025006    L10.cmnd & ~(tty.cr.ten!tty.cr.ren) -> mbr; Disable xmtr/rcvr.
025007    L14.devadr + tty.cmnd -> mar(wio); Write control into the
025010    do.ios -> upc                  ;  2651 Command Register.
025011     temp -> L1.ret1

         cacxdv.on:                      ; Here to restore device state.
025012    L10.cmnd ! tty.cr.re -> mbr    ; Restore Command Register and
025013    L14.devadr + tty.cmnd -> mar(wio);  clear error flags.
025014    do.ios -> upc
025015     (.+1) -> L1.ret1
025016    L10.cmnd -> mbr
025017    L14.devadr + tty.cmnd -> mar(wio)
025020    do.ios -> upc
025021     temp -> L1.ret1

         cacxdv.ena:                     ; Here to take device and start I/O.
025022    l13.inp -> base
025023    l3.iocb ls                     ; Is I/O in progress?
025024    ifnot zero cacxdv.pag -> upc   ; Yes.  Go put away the current IOCB
025025     ifnot zero (.+1) -> l2.ret2   ;  then get another.

025026    l12.dev -> temp                ; Will eventually switch to USYS.
025027    rb.main -> base                ; Fix base for MAIN initially.
025030    base -> g0.temp                ; Restore our environment when done.
025031    14.b & cac.opt.ddt ls          ; Enable DDT?
025032    temp -> base                   ; First switch to USYS.
025033    base -> L0.base                ; Save old base.
025034    cacxdv.off -> upc              ; Go turn off the device.
025035     (.+1) -> temp
025036    if zero L13.ttyflgs ! (tty.own.m!tty.trans) -> L13.ttyflgs; Take con
025037    ifnot zero L13.ttyflgs ! (tty.own.m!tty.secho) -> L13.ttyflgs; with
025040    cacxdv.on -> upc               ; Restore device Command Register and
025041     (.+1) -> temp                 ;  clear status error flags.

025042    L0.base -> base                ; Switch back to MAIN.
025043    14.b & cac.opt.afc ls          ; Automatic flow control?
025044    14.b & cac.opt.hib -> temp     ; First mask off sequence flag.
025045    g0.temp -> base                ; Restore input side environment.
025046    ifnot zero l10.csr ! cac.csr.afc -> l10.csr; Set flow control flag i
                                         ;  required.
025047    temp ls                        ; Hi-Lo byte sequence?
025050    if zero icac.hib -> l14.svc    ; Set up input side UPC for appropria
025051    ifnot zero icac.lob -> l14.svc ;  byte sequence.
025052    l10.csr ! cac.csr.ena -> l10.csr; Indicate device driver is enabled.

025053    l13.inp + caci2o -> base       ; Fix base for output side.
```

```
025054   if zero ocac.lob -> l14.svc     ; Set up output side UPC for appropri
025055   ifnot zero ocac.hib -> l14.svc  ;  byte sequence.

025056   l3.iocb ls                      ; Got an IOCB on output side?
025057   if zero dd.ret -> upc           ; No, so just return.
025060   l10.csr ! cac.csr.ena -> l10.csr ; In either case, indicate device i
025061   ifnot zero cacxdv.pag -> upc    ; Else, go put it away then try to
025062    (.+1) -> l2.ret2               ;  get another IOCB.

         cacxdv.xon:
025063   if zero dd.ret -> upc           ; Return to caller if false.
025064   ifnot zero ocac.1st -> upc      ; Else, start output I/O immediately.
025065    dd.ret -> l1.ret1              ; Return to caller when done.

         cacxdv.pag:
025066   cacxdv.dis -> upc               ; Put away the current IOCB then
025067    cac.start -> temp              ;  try to get another one.

         cacxdv.brk:                     ; Here to send "break" signal.
025070   l11.tmp ls                      ; Input or output side?
025071   if zero dd.ret -> upc           ; If input, leave now.
025072   ifnot zero l3.iocb ls           ; Else check for an active IOCB.
025073   ifnot zero cacxdv.dis -> upc    ; I/O in progress, go discard IOCB.
025074    ifnot zero (.+1) -> temp

025075   l12.dev -> base                 ; Fix base for USYS.
025076   L10.cmnd ! (tty.cr.ten!dev.cmd.brk) -> L10.cmnd,mbr; Set "break" fla
025077   L14.devadr + tty.cmnd -> mar(wio); Write new command word
025100   do.ios -> upc                   ;  into Command Register.
025101    (.+1) -> L1.ret1
025102   dd.ret -> upc                   ; Return to caller with the
025103    L13.ttyflgs ! tty.out -> L13.ttyflgs;  transmitter jammed.


         cacxdv.mrk:                     ; Here to send "mark" signal.
025104   l12.dev -> base
025105   L10.cmnd & ~(tty.cr.ten!dev.cmd.brk) -> L10.cmnd,mbr; Clear "break"
025106   L14.devadr + tty.cmnd -> mar(wio); Write new command word.
025107   do.ios -> upc                   ; [1-6]
025110    dd.ret -> L1.ret1              ; Return eventually.


         cacxdv.bps:                     ; Here to set baud rate.
025111   rb.main -> base                 ; Fix base to access B-register.
025112   base -> g0.temp                 ; Save base.

025113   l4.b ls                         ; Index should be positive.
025114   if neg crash -> upc             ; Bad index.
025115    if neg cacerr.ibrr -> temp     ; ++ Illegal Baud Rate Request ++

025116   l4.b - cacbps.len ls            ; Index in range?
025117   ifnot neg crash -> upc          ; If not, issue error msg.
025120    ifnot neg cacerr.ibrr -> temp

025121   cacbps.tbl -> temp              ; Calculate table index.
025122   l4.b + temp -> upc              ; Put it in safe place.
025123   cacbps.set -> upc
```

```
                cacbps.tbl:
025124            baud.110 -> temp
025125            baud.300 -> temp
025126            baud.1200 -> temp
025127            baud.1800 -> temp
025130            baud.2400 -> temp
025131            baud.4800 -> temp
025132            baud.9600 -> temp
025133            baud.19200 -> temp
000010          cacbps.len = (. - cacbps.tbl)  ; Baud rate table length.

                cacbps.set:
025134          g0.temp -> base                ; Restore base.
025135          l12.dev -> base                ; Fix base.
025136          L10.cmnd & ~(tty.cr.ten!tty.cr.ren) -> mbr; Turn off device.
025137          L14.devadr + tty.cmnd -> mar(wio); Write Command Register.
025140          do.ios -> upc
025141            (.+1) -> L1.ret1
025142          mar -> mar(rio)                ; Read Command Register so that
025143          do.ios -> upc                  ;  we're certain Mode Register 1
025144            (.+1) -> L1.ret1             ;  (MR1) is accessed first.

025145          setup.2651 -> mbr              ; Write same configuration into MR1.
025146          L14.devadr + tty.mr -> mar(wio); (setup.2651 is defined in USYS as
025147          do.ios -> upc                  ;  Asynch, 16X rate, 1 stop bit)
025150            (.+1) -> L1.ret1
025151          temp -> mbr                    ; Now write baud rate into MR2.
025152          do.ios -> upc
025153            (.+1) -> L1.ret1

025154          L10.cmnd ! tty.cr.re -> mbr    ; Clear error flags.
025155          L14.devadr + tty.cmnd -> mar(wio);
025156          do.ios -> upc
025157            (.+1) -> L1.ret1
025160          L10.cmnd -> mbr                ; Restore original command word.
025161          L14.devadr + tty.cmnd -> mar(wio)
025162          do.ios -> upc
025163            dd.ret -> L1.ret1            ; Dismiss when done.


                cacxdv.xfr:                    ; Here to service single transfers.
025164            l10.csr & cac.csr.ena ls      ; Have we been enabled?
025165            ifnot zero cacxdv.ok -> upc  ; If so, continue as usual.
025166            if zero cacxdv.ans -> upc    ; Otherwise, return an
025167              -1 -> temp                 ;  error response.

                cacxdv.ok:
025170            l11.tmp ls                   ; Is it an input or output transfer?
025171            ifnot zero cacxdv.put -> upc ; If output, go do a "put char."
025172              ifnot zero l10.csr & cac.csr.out ls; Check if output is being attem

025173            l10.csr & cac.csr.xfr ls      ; Provide status of current transfer
025174            if zero dd.ret -> upc        ; If no input has been received retur
025175            ifnot zero l3.iocb ls        ; Else check for an active IOCB?
025176            ifnot zero dd.ret -> upc     ; I/O in progress, return now to indi
                                               ;  that device is busy.
025177            if zero l12.dev -> base      ; No current I/O, so switch to USYS.
025200            base -> L0.base              ; Save our environment.
```

```
025201   L5.in.stat & dev.sts.err ls    ; Did device malfunction?
025202   L0.base -> base                ; Switch back to our device driver.

025203   l6.char -> temp                ; Get the saved character.
025204   l10.csr & ~cac.csr.xfr -> l10.csr; Clear "transfer" flag.
025205   if zero l10.csr & cac.csr.ovr ls; If no device errors, check for
                                        ;  multiple transfers.
025206   rb.main -> base                ; Fix base to access B-reg.
025207   temp -> 14.b                   ; Load character into B-reg.
025210   if zero 0 -> 13.a              ; If no errors, indicate success.
025211   dd.ret -> upc                  ; Return to caller.
025212    ifnot zero -1 -> 13.a         ; Error detected, respond accordingly


         cacxdv.ans:
025213   rb.main -> base                ; Fix base.
025214   dd.ret -> upc                  ; Return to caller with response
025215    temp -> 13.a                  ;  in accumulator.


         cacxdv.put:                    ; Here to write a byte.
025216   ifnot zero dd.ret -> upc       ; Output buffer is full so return
                                        ;  to indicate that we are busy.
025217   if zero l10.csr & (cac.csr.xfr!cac.csr.afc) ls; Else, check whether
                                        ;  DEVICE is busy or flow control is
025220   l12.dev -> temp                ; First, get pointer to handler in US
025221   rb.main -> base                ; Next, switch to Main to get char.
025222   base -> g0.temp                ; Save old base.
025223   14.b -> mbr                    ; Get the character.
025224   if zero cacxdv.xmt -> upc      ; May attempt character transfer.
025225   ifnot zero g0.temp -> base     ; Can't attempt transfer, restore bas
025226   mbr -> l16.buf                 ; Save character.
025227   l10.csr ! cac.csr.out -> l10.csr; Set "buffer full" flag.
025230   cacxdv.ans -> upc              ; Give caller access response.
025231    0 -> temp                     ; Indicate "success."

         cacxdv.xmt:
025232   temp -> base                   ; Switch to device handler in USYS.
025233   base -> L0.base                ; Save old base which points to MAIN.
025234   L13.ttyflgs & tty.own.m ls     ; Is device in our control?
025235   if zero dd.ret -> upc          ; No, return now indicating device is
025236   ifnot zero mbr -> temp         ; Else, get the character then
025237   ttyxmit -> upc                 ;  go transmit it.
025240    (.+1) -> L3.ret3              ; Come back to check results.

025241   L13.ttyflgs & tty.out ls       ; Did transmitter take it?
025242   ifnot zero g0.temp -> base     ; If not, restore base.
025243   ifnot zero l10.csr ! cac.csr.xfr -> l10.csr; Set "transfer" flag.
025244   cacxdv.ans -> upc              ; Go give caller access response.
025245    0 -> temp                     ; Indicate "success."


         cacxdv.abt:                    ; Here to discard current IOCB.
025246   l3.iocb ls                     ; Active IOCB?
025247   if zero dd.ret -> upc          ; No, so just return (a no-op).
025250   ifnot zero cacxdv.pag -> upc   ; Else, discard IOCB then try
025251    (.+1) -> l2.ret2              ;  for another.
025252   if zero dd.ret -> upc          ; Didn't get one, so return.
```

```
025253   ifnot zero cacxdv.xon -> upc    ; Otherwise, go check if I/O
025254    111.tmp ls                     ;  must start immediately.

         cacxdv.sts:                    ; Here to read status.
025255   .110.csr -> temp                 ; Provide device driver status.
025256   112.dev -> base                 ; Switch to USYS.
025257   L14.devadr + tty.sts -> mar(rio); Read device status.
025260   do.ios -> upc
025261    (.+1) -> L1.ret1

025262   mbr -> g0.temp                  ; Copy device status.
025263   g0.temp & msk377 -> g0.temp     ; Extract significant (low-order) byt
025264   cacxdv.ans -> upc               ; Go return response.
025265    g0.temp ! temp -> temp         ; Form hardware/software status word.


         cacxdv.cle:                    ; Here to clear hard/soft error flags
025266   113.inp -> base
025267   110.csr & ~(cac.csr.xfr!cac.csr.ovr!cac.csr.err) -> 110.csr;Clear fl
025270   112.dev -> base                 ; Switch to USYS.
025271   cacxdv.on -> upc                ; Restore device command word and cle
025272    dd.ret -> temp                 ;  its error flags.  Return when done
```

```
        ; PDV, Microcall: poke input device.

        cac.pdv:
025273   10.dcb -> temp                    ; Copy DCB back-pointer.
025274   g6.temp - temp ls                 ; Is it our DCB?
025275   ifnot zero crash -> upc           ; No, so issue error msg.
025276    ifnot zero ioerr.rbmu -> temp    ; ++ Register Block Messed Up ++

025277   110.csr & cac.csr.ena ls           ; Do we have control of device?
025300   if zero crash -> upc              ; If not, issue error message.
025301    if zero cacerr.nena -> temp      ; ++ Not Enabled ++

025302   13.iocb ls                        ; Active IOCB?
025303   if zero cacpdv.io -> upc          ; No, go attempt to start I/O.
025304   ifnot zero dd.ret -> upc          ; If there is, return now.
025305    nop

        cacpdv.io:                        ; Here to initiate I/O.
025306   cac.start -> upc                  ; Try to get an IOCB.
025307    (.+1) -> 12.ret2
025310   if zero dd.ret -> upc             ; No IOCB was retrieved, return now.
025311   ifnot zero 115.bgn -> upc         ; Else, commence with I/O and check
025312    110.csr & cac.csr.xfr ls          ;  for a transfer in progress.


        ; START Subroutine.

        cac.start:
025313   10.dcb -> temp                    ; Get process identifier then try to
025314   iodd.get -> upc                   ;  get and IOCB.
025315   (.+1) -> g6.temp

025316   if zero 12.ret2 -> upc            ; Didn't get one, so leave now.
025317   ifnot zero g0.temp -> temp        ; Put IOCB ptr in safe place
025320   temp -> 13.iocb                   ;  then save it in local register.

025321   13.iocb + iocb.addr -> mar(r)     ; Read the buffer address.
025322   nop                               ; [1]
025323   nop                               ; [2]
025324   mbr -> 14.addr                    ; Save it in local register.

025325   13.iocb + iocb.size -> mar(r)     ; Read the buffer size.
025326   nop                               ; [1]
025327   nop                               ; [2]
025330   mbr -> 15.left ls                 ; Also save it locally.
025331   if zero crash -> upc              ; No buffer space was allocated.
025332    if zero ioerr.ilts -> temp       ; ++ Illegal Transfer Size ++

025333   15.left & 7 ls                    ; Is size a multiple of eight?
025334   if zero 12.ret2 -> upc            ; It is, so return now and give
025335    if zero 13.iocb ls               ;  indication of an active IOCB.

025336   crash -> upc                      ; Specified size not multiple of eigh
025337    ioerr.ilts -> temp
```

```
        ; CAS.SCC -Set Completion Code Subroutine

        cac.scc:
025340   13.iocb + iocb.flags -> mar(r) ; Get IOCB flags.
025341   nop                           ; [1]
025342   nop                           ; [2]
025343   mbr ! g0.temp -> mbr          ; Set completion code.

025344   13.iocb + iocb.flags -> mar(w) ; Write back flags.
025345   nop                           ; [1]
025346   nop                           ; [2]

        ; Fallthrough into completion routine.


        ; CAS.DONE Subroutine.

        cac.done:
025347   13.iocb + iocb.size -> mar(r)  ; Get buffer size in order to compute
02535C   nop                           ; [1] the total number of bytes trans
025351   0 -> g0.temp                  ; [2] Need a zero in subsequent compu

025352   15.left - mbr -> mbr          ; |MBR| = # of bits transfered.
025353   g0.temp - mbr -> mbr          ; Convert to positive number.
025354   13.iocb + iocb.size -> mar(w)  ; Write transfer count into IOCB.
025355   10.dcb -> temp                ; [1] Provide process identifier.
025356   iodd.put -> upc               ; [2] Go put away IOCB.(Return link i
025357    0 -> 13.iocb                 ; There is no longer an active IOCB.

02536C  unext = .
        # 22 "m.mic"
```

```
# 1 "cm1.mic"
```

```
        ;Cheap MII 2651 driver (both in & out)

        ;Known bugs/deficiencies:

        ;CM10 SYN logic sends SYN-frame-SYN even if the line was idle
        ; ... for a long time.
        ;CM1, unlike C18 will cause interrupts on XDV resets because
        ; ... it's not smart about calling IODD.NPUT in those cases. See C18,

        ;Most symbols are prefixed with "CM1" (read cheap MII 2651). Those
        ;prefixed with "CM1I" apply to the input side only, those with "CM10"
        ;to the output side.

        ;Completion codes:

000001  cm1.cc.error  == 1                ;error completion code

        ;XDV functions supported by both the input and output sides.

        ;(0) No-op
        ;(1) Enable 2651 and clear software
        ;(2) Disable 2651 and clear software
        ;(3) loop interface
        ;(4) loop modem
        ;(5) unloop both interface and modem, reset and enable
```

```
        ;Register definitions. "i" = input, "o" = output, "b" = both

000020  cm1.in2out == 20               ;offset from input to output RB

000000  10.dcb     == 10               ;(b) pointer back to DCB
000003   13.lobyte == 13               ;(i) lo byte buffer, -1=none
000003   13.hibyte == 13               ;(o) hi byte buffer, -1=none
000003   13.count  == 13               ;(b) ... also doubles as CRC counter
000004  14.iocb    == 14               ;(b) pointer to IOCB, 0=none
000005  15.addr    == 15               ;(b) next word to go out
000006  16.left    == 16               ;(b) number of words left to output
000007  17.next    == 17               ;(b) UPC to use to service next chara
000010  110.crca   == 110              ;(b) CRC computation registers
000011  111.crcb   == 111
000012  112.crcc   == 112
000013  113.crcd   == 113
000014  114.cr     == 114              ;(i) pointed to by L16 of both (i) an
                                      ;(o) unused
000015  115.usart  == 115              ;(b) USART IO address associated w/ t
000016  116.crptr  == 116              ;(b) pointer to register containing C
000017  117.char   == 117              ;(b) holds current character (occaisi
```

```
        ;Configuration

000001  cm1i      == 1                ;device number for input
000002  cm1o      == 2                ;device number for output

        ;2651 Definitions

000000  u2651.rx  == 0                ;receive data register in 2651 (read-
000000  u2651.tx  == 0                ;transmit data register (write-only)
000001  u2651.sr  == 1                ;status register (read-only)
000001  u2651.ssd == 1                ;SYN-1, SYN-2, DLE register (write-on
000002  u2651.mr  == 2                ;mode registers #1 & #2 (read-write)
000003  u2651.cr  == 3                ;control regster (read-write)
000200  u2651.mcr == 200              ;modem control register &&?

000200  u2651.sr.dtr     == 200       ;DTR
000100  u2651.sr.dcd     == 100       ;DCD
000040  u2651.sr.syn     == 040       ;detected SYN
000020  u2651.sr.ovr     == 020       ;receiver overrun
000002  u2651.sr.rxrdy   == 002       ;receiver has character
000001  u2651.sr.txrdy   == 001       ;transmitter wants character

000040  u2651.cr.rts     == 040       ;asserts RTS (causing LOOP on ARPAnet
000020  u2651.cr.reset   == 020       ;resets receiver
000004  u2651.cr.rxen    == 004       ;enables receiver
000002  u2651.cr.dtr     == 002       ;asserts DTR
000001  u2651.cr.txen    == 001       ;enables transmitter

000214  u2651.sync.mr1   == 214       ;Mode register #1 for synchronous ope
000000  u2651.sync.mr2   == 000       ; . . . . . #2 . . . . . .
000005  u2651.sync.cr    == 005       ;Control register . . . . . .

000316  u2651.xp.mr1     == 316       ;Mode register #1 for crosspatched op
000077  u2651.xp.mr2     == 077       ; . . . . . #2 . . . . . .
000243  u2651.xp.cr      == 243       ;Control register . . . . . .

        ;CSR definitions

000001  cm1csr.rxien     == 1         ;receiver interrupt enable
000002  cm1csr.txien     == 2         ;transmitter interrupt enable
000004  cm1csr.enable    == 4         ;receiver/transmitter enable (on-off)
000010  cm1csr.bit3      == 10        ; ** reserved
000007  cm1csr.enall     == (cm1csr.rxien ! cm1csr.txien ! cm1csr.enable)

        ;Character codes, ODD parity

000002  cm1.stx          == 002       ;start of text (frame)
000203  cm1.etx          == 203       ;end of text (frame)
000020  cm1.dle          == 020       ;escape character
000026  cm1.syn          == 026       ;idle line synchronization character
```

```
025360   uram unext

         cm1i.0:                          ;function dispatch - input side
025360    cm1i.apr -> upc
025361    cm1.dpr -> upc
025362    cm1.xdv -> upc
025363    cm1.pdv -> upc

         cm1o.0:
025364    cm1o.apr -> upc
025365    cm1.dpr -> upc
025366    cm1.xdv -> upc
025367    cm1.pdv -> upc
```

```
                ;Inputerrupt level - input side

                cm1i.int:                               ;here on microinterupts, BASE points
                                                        ;RB.CM1I.n where n=unit number (0-5)
025370           l15.usart + u2651.rx -> mar(rio) ;read the character, latch L15.USAR
025371           m2.ios -> misc2                        ;like DO.IOS
025372           m2.ios -> misc2
025373           m2.ios -> misc2
025374           377 -> l17.char                        ;prepare for reading character down t
025375           l17.char & mbr -> l17.char
025376           l7.next -> temp ls                     ;going somewhere?
025377           ifnot zero cm1i.going -> upc           ;yes, check for IOCB, etc.
025400           if zero l17.char - ascii.dle ls ;would be going to NEW, is this a DL
025401           ifnot zero io.ret -> upc               ;nope, skip it
025402           if zero l4.iocb ls                     ;we have an IOCB to handle the DLE?
025403           ifnot zero cm1i.new -> upc             ;yes, go to NEW
025404           if zero io.ret -> upc                  ;no, just return

                cm1i.going:
025405           l15.usart ls                           ;is this a good interrupt?
025406           ifnot zero cm1.crc -> upc              ;if so, update CRC
025407           ifnot zero temp -> l1.ret1             ;then dispatch through L7.NEXT
025410           if zero crash -> upc                   ;if we get an interrupt without L15 s
025411           if zero ioerr.uint -> temp             ; ++ unexpected interrupt
```

```
        ;Here on the DLE before a packet.

        cm1i.new:                           ;here before first character
025412   0 -> l10.crca                      ;since STX is included in CRC, prime
025413   0 -> l11.crcb                      ;so STX will factor in if it's truly
025414   0 -> l12.crcc                      ;we get on the line
025415   cm1i.stx -> l7.next                ;and look for STX next
025416   cm1.crc -> upc                     ;update the CRC to include the DLE
025417   io.ret -> l1.ret1

        cm1i.stx:                           ;here after first DLE
025420   l17.char - cm1.stx ls              ;an STX?
025421   ifnot zero cm1.resync -> upc       ;if not, re-sync

        ;Set L3.LOBYTE to expect low byte first and process data bytes which

025422   cm1i.data -> l7.next               ;process data bytes form now on
025423   io.ret -> upc                      ;return and dismiss directly
025424   -1 -> l3.lobyte                    ;starting with the low byte first

        ;Here when the receiver has run into trouble and is in need of
        ;re-synchronization. Clear the 2651 receiver and call CM1.RESTART
        ;to re-start the transfer. Since we can be called from either the
        ;input block or the output block, we must locate the pointer to
        ;the CSR using L16.CRPTR.

        cm1.resync:
025425   l16.crptr -> base                  ;make L0 point to the CSR word
025426   base -> g1.temp                    ;save old base

025427   l0 & ~u2651.cr.rxen -> mbr         ;turn off receiver first..
025430   g1.temp -> base                    ;fix base
025431   l15.usart + u2651.cr -> mar(wio)
025432   do.ios -> upc
025433   (.+1) -> l1.ret1

025434   l16.crptr -> base
025435   l0 ! u2651.cr.reset -> mbr         ; ... then reset and turn on
025436   g1.temp -> base
025437   l15.usart + u2651.cr -> mar(wio)
025440   do.ios -> upc
025441   (.+1) -> l1.ret1

025442   cm1.restart -> upc                 ;restart the current io
025443   io.ret -> l1.ret1                  ; ... setting L4.NEXT to 0 as well
```

```
        ;Here on each sucessive data byte.

        cm1i.data:
 025444  l17.char - cm1.dle ls          ;a DLE?
 025445  if zero io.ret -> upc          ;yes
 025446  if zero cm1i.dle -> l7.next    ; process next character elsewhere

        ;The bytes coming in are low byte, high byte, low byte, etc. Thus, we
        ;must buffer the low byte until we get the corresponding high byte
        ;at which time we write 16 bits into macromemory. The register L3.LOB
        ;is that buffer. If L3.LOBYTE is negative (nominally -1), there is no
        ;buffered low byte. In this case, we copy the current character into
        ;thus making it positive. When it was previously positive, we concate
        ;its contents with the incoming (high) byte (shifted, of course) and
        ;the word, then we set LOBYTE to -1 again.

        cm1i.store:                     ;(enter here from CM1I.DLE when DLE D
 025447  l3.lobyte ls                   ;check the low byte flag/buffer
 02545C  ifnot neg cm1i.high -> upc     ;if it's not negative, L17.CHAR is th
 025451  ifnot neg l17.char -> mbr      ; ... so set up MBR for swapping shor

 025452  l17.char -> temp               ;copy this char to become low byte
 025453  io.ret -> upc                  ;done
 025454  temp -> l3.lobyte

        ;Here for the high byte (in L17.CHAR and MBR). Low byte buffered in L

        cm1i.high:
 025455  l6.left - 20 -> l6.left ls          ;decrment count and check
 025456  ifnot neg l3.lobyte ! s16mbr -> mbr  ;set up to write the word to
 025457  ifnot neg l5.addr -> mar(w)          ;write the word to macromemor
 02546C  ifnot neg l5.addr + 1 -> l5.addr     ;[1] increment pointer
 025461  ifnot neg io.ret -> upc              ;[2] if count ok, return to d
 025462  ifnot neg -1 -> l3.lobyte            ;next thing in will be low by

        ;The IOCB is exhausted.  Call CM1.EDONE to error-out the anemic IOCB
        ;favor of a new one.

 025463  l6.left + 20 -> l6.left             ;compensate for extra decrem

        ;Here when the count went to zero before the input stopped. Call EDON
        ;to error out this IOCB and get a new one. Also used as general purpo
        ;error and return vector.

        cm1.error:                      ;(enter here from all over)
 025464  cm1.edone -> upc
 025465  io.ret -> l2.ret2              ;clear up, return when done
```

```
        ;Here on DLE's in the data stream. DLE DLE begets a true DLE fed into
        ;data. DLE SYN is ignored. DLE ETX signals the end of frame, CRC to f

        cm1i.dle:
025466   cm1i.data -> 17.next           ;assume we'll return to data
025467   117.char - cm1.dle ls          ;a second DLE?
025470   if zero cm1i.store -> upc      ;yes, use it as real data

025471   ifnot zero 117.char - cm1.syn ls ;a SYN?
025472   if zero io.ret -> upc          ;yes, ignore DLE SYN

025473   117.char - cm1.etx ls          ;an ETX?
025474   ifnot zero cm1.resync -> upc   ;no, resync

        ;Here when we are at the end of the frame (with dangling UPC). Make s
        ;there is no dangling low byte (if there is, call EDONE instead). Fin
        ;setup to check the CRC next.

025475   13.lobyte ls                   ;any low byte?
025476   ifnot neg cm1.error -> upc     ;if so, handle the error condition el
025477   if neg cm1i.checkcrc -> 17.next ;set up to handle the CRC next
025500   io.ret -> upc                  ;return
025501    3 -> 13.count                 ;set up the CRC counter to 3 bytes to
```

```
        ;Here to process CRC bytes. L3.CRCCNT counts from 3 to zero while we
        ;accept the CRC from the line and accumulate it with itself. When thi
        ;is complete, the CRC accumulators (L10.CRCa,b,c) should be zero.

        cm1i.checkcrc:
 025502  l3.count - 1 -> l3.count ls     ;are we done yet?
 025503  ifnot zero io.ret -> upc        ;if not, dismiss until we are

        ;Check the CRC accumulators for zero

 025504  if zero l10.crca ls             ;(dangling IFNOT)
 025505  if zero l11.crcb ls             ;if CRCA was zero, test B
 025506  if zero l12.crcc ls             ;if CRCA and B were zero, test C

 025507  ifnot zero cm1.error -> upc     ;if they don't check to zero, error

        ;else check for USART overrun

 025510  if zero l15.usart + u2651.sr -> mar(rio)
 025511  do.ios -> upc
 025512  (.+1) -> l1.ret1
 025513  mbr -> l17.char                 ;look at status register
 025514  l17.char & u2651.sr.ovr ls      ;overrun?
 025515  ifnot zero cm1.error -> upc     ;if so, error this one out

        ;all was well, we're done with this IOCB

 025516  if zero cm1.done -> upc         ;call DONE
 025517  io.ret -> l2.ret2               ;return and dismiss
```

```
        ;Interrupt elvel - output side

        cm1o.int:

025520   115.usart ls                    ;make sure this isn't a bogus interru
025521   if zero crash -> upc
025522   if zero ioerr.uint -> temp
025523   14.iccb ls                      ;is there a current IOCB?
025524   if zero cm1o.idle -> upc        ;if not, just send SYNs
025525   ifnot zero 17.next ls           ;if there is, dispatch
025526   ifnot zero 17.next -> upc       ;if L7.NEXT=0, goto CM1O.NEW
025527   if zero cm1o.new -> 17.next, upc
025530   nop

        ;Here when there's no current IOCB, simply output a SYN

        cm1o.idle:
025531   cm1.syn -> mbr                  ;get a SYN
025532   115.usart + u2651.tx -> mar(wio)
025533   do.ios -> upc                   ;output it
025534   (.+1) -> 11.ret1                ;[1,2]
025535   nop                             ;[3]
025536   nop                             ;[4]
025537   nop                             ;[5]
025540   io.ret -> upc                   ;[6] dismiss
025541   nop                             ;[7]
```

```
        ;Here at the start of a new IOCB for output. Send one SYN now to guar
        ;at least 2 SYNs between frames (the second SYN is sent after the CRC

        cm1o.new:
025542   cm1o.idle -> upc               ;merge with SYN sender
025543   cm1o.sop -> 17.next            ;come back here when done

        cm1o.sop:
025544   0 -> 110.crca                  ;clear the CRC
025545   0 -> 111.crcb
025546   0 -> 112.crcc

025547   cm1.dle -> 117.char            ;send a DLE
025550   cm1o.send1 -> upc
025551   (.+1) -> 17.next

025552   cm1.stx -> 117.char            ;then an STX
025553   -1 -> 13.hibyte                ;say we need to send the low byte
025554   cm1o.send1 -> upc
025555   cm1o.data -> 17.next           ;then send data
```

```
        ;Here on each transmit interrupt. First check to see if we have buffe
        ;a high byte in L3.HIBYTE which needs to go out.

        cm1o.data:
025556   l3.hibyte -> temp ls              ;need to output the byte?
025557   if neg cm1o.low -> upc            ;if not, try for new low byte in new
025560   -1 -> l3.hibyte                   ;remove the hi byte we're done with
025561   cm1o.send -> upc                  ;send it, doing DLE doulbing, etc.
025562   temp -> l17.char                  ;arg to CM10.SEND, etc.

        ;Here on each new word (low byte)

        cm1o.low:
025563   l6.left - 20 -> l6.left ls        ;anything more to fetch?
025564   if neg cm1o.eop -> upc            ;if not, end packet
025565   ifnot neg l5.addr -> mar(r)       ;else fetch fetch word
025566   l5.addr + 1 -> l5.addr            ;[1] step to next
025567   377 -> l17.char                   ;[2] low 8 bits only!
025570   s16mbr -> l3.hibyte               ;set up the hi byte
025571   l3.hibyte & 377 -> l3.hibyte      ;low 8 bits (better not be negative!)
025572   l17.char & mbr -> l17.char        ;output this byte

        ;Enter here to output a byte to the line from L17.CHAR. Knows how to
        ;DLE doubling.

        cm1o.send:
025573   l17.char - cm1.dle ls             ;is this a DLE?
025574   if zero cm1o.dle -> l7.next       ;if so, set up to output DLE again sh

        cm1o.send1:                       ;(enter here from CM10.DLE, below)
025575   l17.char -> mbr                   ;move char to useful place
025576   l15.usart + u2651.tx -> mar(wio)
025577   do.ios -> upc                     ;output it
025600   (.+1) -> l1.ret1
025601   cm1.crc -> upc                    ;update the checksum
025602   io.ret -> l1.ret1                 ;return and dismiss

        ;Here when we've just output a DLE and need to output another.

        cm1o.dle:
025603   cm1.dle -> l17.char               ;set up a DLE
025604   cm1o.send1 -> upc                 ;output it (don't check for DLE, tho!
025605   cm1o.data -> l7.next              ;go back to normal DATA loop
```

```
        ;Here on end of packet. Send DLE ETX crc SYN

        cm1o.eop:
025606   16.left + 20 -> 16.left          ;compensate for extra bump
025607   cm1.dle -> 117.char
025610   cm1o.send1 -> upc                ;send it
025611   (.+1) -> 17.next

025612   cm1.etx -> 117.char
025613   3 -> 13.count                    ;set up CRC counter
025614   cm1o.send1 -> upc
025615   cm1o.crc -> 17.next

        ;Here to transmit a CRC byte, L3.COUNT has counter from 3 to 0.

        cm1o.crc:
025616   0 -> 117.char                    ;arg of zero means read it back
025617   cm1.crc -> upc
025620   (.+1) -> 11.ret1
025621   0 -> 110.crca                    ;(see comment at CM1.CRC)

025622   g0.temp -> mbr                   ;output the CRC byte ourselves
025623   115.usart + u2651.tx -> mar(wio)
025624   do.ios -> upc
025625   (.+1) -> 11.ret1
025626   nop                              ;[3]
025627   13.count - 1 -> 13.count ls      ;[4] update count
025630   if zero cm1o.crcend -> 17.next ;[5] if all done, don't return here
025631   io.ret -> upc                    ;[6] return and do next CRC later
025632   nop                              ;[7]

        ;Here after all 3 CRCs have been sent, send a SYN

        cm1o.crcend:                     ;(from above after all 3 CRC bytes ou
025633   cm1o.idle -> upc                ;send a SYN
025634   (.+1) -> 17.next

        ;After sending SYN, finish up

025635   cm1.done -> upc                  ;send the packet
025636   io.ret -> 12.ret2                ;return and dismiss - next time aroun
                                          ;L4.IOCB will be zero and we'll go di
                                          ;to CM10.IDLE, or L7.NEXT will be zer
                                          ;go directly to CM10.NEW.
```

```
        ;CM1.CRC, Subroutine, Level 1

        ;Computes the 24 bit CRC associated with ARPAnet trunk lines. On inpu
        ;L17.CHAR has the current character to be included. Call with L17.CHA
        ;to read next CRC byte for output into G0.TEMP. In this case, caller
        ;clear L10.CRCA.

        cm1.crc:
        cm1.crc.patc:
025637   l12.crcc + 0 -> mar(rp)          ;fetch table entry for CRCc
02564C   l17.char -> temp                 ;[1] move arg to CRCD
025641   temp -> l13.crcd                 ;[2]

        cm1.crc.patb:
025642   l11.crcb + 0 -> mar(rp)
025643   mbr -> g0.temp                   ;[1] catch CRCTAB.C value before it v
025644   nop                              ;[2]

        cm1.crc.pata:
025645   l10.crca + 0 -> mar(rp)
025646   g0.temp ? mbr -> g0.temp         ;[1] pick up contributions from CRCTA
025647   nop                              ;[2]

02565C   g0.temp ? mbr -> g0.temp, temp
025651   l13.crcd ? temp -> l13.crcd

        cm1.crc.patd:
025652   l13.crcd + 0 -> mar(rp)
025653   l11.crcb -> temp                 ;[1] shuffle CRCb and CRCa right
025654   temp -> l12.crcc                 ;[2]
025655   l10.crca -> temp
025656   temp -> l11.crcb

025657   l1.ret1 -> upc                   ;return
02566C   mbr -> l10.crca                  ; ... picking up CRCTAB.D's entry in
```

```
        ;Here on APRs to a CM1. Clear the interface and enable the 2651, then
        ;try to start io going. Enter with BASE=DCB.RB from macrocode and G6.
        ;pointing to the DCB we're APR'ing.

        cm1i.apr:
025661   g6.temp -> temp                ;set up back pointer to DCB
025662   temp -> 10.dcb                 ;set back pointer
025663   cm1.reset -> upc
025664   cm1.xdv.ena -> 12.ret2         ;reset both sides

        cm1o.apr:
025665   g6.temp -> temp
025666   dd.ret -> upc
025667   temp -> 10.dcb
```

```
        ;Here with standard args (G6=DCB, BASE=base) on a DPR of the process

        cm1.dpr:
025670   10.dcb -> temp                  ;make sure we point back to the DCB
025671   g6.temp - temp ls
025672   ifnot zero crash -> upc         ;if they don't match, bomb
025673   ifnot zero ioerr.rbmu -> temp   ; ++ Register blocks messed up

025674   14.iocb ls                      ;any current IOCB?
025675   ifnot zero cm1.edonestop -> upc ;yes, error it out w/o getting new o
025676   ifnot zero (.+1) -> 12.ret2
025677   14.iocb ls                      ;make sure we got rid of it (&&)
025700   ifnot zero crash -> upc         ;if it isn't...
025701   ifnot zero ioerr.csii -> temp   ; ++ cant shake IOCB when IDLE

025702   0 -> mbr                        ;turn off hardware
025703   115.usart + u2651.mcr -> mar(wio)
025704   do.ios -> upc
025705   dd.ret -> 11.ret1
```

```
        ;Here on XDV. BASE and G6 as usual. G0 contains user's AC (see
        ;I0316.ACT).

        cm1.xdv:
025706   cm1.xdv.table -> temp
025707   iodd.xdv -> upc
025710   cm1.xdv.size -> mar

        cm1.xdv.table:
025711   dd.ret -> upc                  ;(0) no-op
025712   cm1.xdv.ena -> upc             ;(1) enable and reset
025713   cm1.xdv.dis -> upc             ;(2) disable and reset
025714   cm1.xdv.loopint -> upc         ;(3) loop interface
025715   cm1.xdv.loopmod -> upc         ;(4) loop modem
025716   cm1.xdv.unloop -> upc          ;(5) unloop both, reset and enable
000006  cm1.xdv.size == (.-cm1.xdv.table)
```

```
                cm1.xdv.unloop:                 ;here on unloop and reset
                cm1.xdv.ena:                    ;here on enable and reset, also from
025717           cm1.zusart -> upc              ;clear the 2651
025720           (.+1) -> l2.ret2
025721           14.iocb ls                     ;an IOCB?
025722           if zero dd.ret -> upc          ;no, return now
025723           ifnot zero cm1.edone -> upc    ;yes, error flush it
025724           dd.ret -> l2.ret2              ;then return to caller

                cm1.xdv.dis:                    ;here on disable and reset
025725           cm1.zusart -> upc              ;clear the 2651
025726           (.+1) -> l2.ret2
025727           14.iocb ls                     ;an IOCB?
025730           if zero dd.ret -> upc          ;no, return now
025731           ifnot zero cm1.edonestop -> upc ;yes, flush it w/o getting new one
025732           ifnot zero dd.ret -> l2.ret2   ;return directly to caller

                cm1.xdv.loopint:                ;here on loop interface
025733           cm1.zusart -> upc              ;first clear 2651
025734           (.+1) -> l2.ret2
025735           u2651.xp.mr1 -> mbr            ;then set up XP registers
025736           115.usart + u2651.mr -> mar(wio)
025737           do.ios -> upc
025740           (.+1) -> l1.ret1

025741           u2651.xp.mr2 -> mbr
025742           115.usart + u2651.mr -> mar(wio)
025743           do.ios -> upc
025744           (.+1) -> l1.ret1

025745           cm1.syn -> mbr
025746           115.usart + u2651.ssd -> mar(wio)
025747           do.ios -> upc
025750           (.+1) -> l1.ret1

025751           116.crptr -> base              ;find correct base
025752           base -> g0.temp                ;remember old one

025753           u2651.xp.cr -> 10, mbr
025754           g0.temp -> base                ;fix BASE
025755           115.usart + u2651.cr -> mar(wio)
025756           do.ios -> upc
025757           (.+1) -> l1.ret1

025760           14.iocb ls                     ;an IOCB?
025761           if zero dd.ret -> upc          ;no, return now
025762           ifnot zero cm1.edone -> upc    ;yes, flush it
025763           ifnot zero dd.ret -> l2.ret2   ;return directly to caller

                cm1.xdv.loopmod:                ;here on loop modem
025764           116.crptr -> base
025765           base -> g1.temp
025766           10 ! u2651.cr.rts -> 10, mbr   ;set RTS to loop an ARPAnet modem
025767           g1.temp -> base
025770           115.usart + u2651.cr -> mar(wio)
025771           do.ios -> upc
025772           (.+1) -> l1.ret1
```

```
025773   14.iocb ls                      ;an IOCB?
025774   if zero dd.ret -> upc           ;no, return now
025775   ifnot zero cm1.edone -> upc     ;yes, flush it
025776   ifnot zero dd.ret -> 12.ret2    ;return directly to caller
```

```
        ;Here on PDVs

        cm1.pdv:
025777   10.dcb -> temp                    ;make sure back pointer is right
026000   g6.temp - temp 1s
026001   ifnot zero crash -> upc
026002   ifnot zero ioerr.rbmu -> temp

026003   cm1.start -> upc                  ;try to start new IO
026004   dd.ret -> 12.ret2
```

```
        ;CM1.RESET, Subroutine, Level 2

        ;Subroutine to master clear a 2651.
        ;Enter with BASE pointing to the register block associated with the i
        ;side. Resets and turns off the device.

        cm1.reset:
        ; 0 -> 10.dcb                          ;clear back pointer to DCB to force
 026005  0 -> 14.iocb                         ;no current IOCB
 026006  0 -> 17.next                         ;encourage a crash if use co-routine

 026007  base -> g2.temp
 026010  g2.temp + cm1.in2out -> base         ;switch to output side

 026011  0 -> 10.dcb
 026012  0 -> 14.iocb
 026013  0 -> 17.next

 026014  g2.temp -> base                      ;restore base

 026015  0 -> mbr                            ;disable the device
 026016  115.usart + u2651.mcr -> mar(wio)
 026017  do.ios -> upc
 026020  (.+1) -> 11.ret1
 026021  12.ret2 -> upc                       ;return
 026022  nop                                  ; ... delay to make strobe at least 1

        ;CM1.ZUSART, Subroutine, Level 2

        ;Clears the 2651's internal registers and CSR, thus removing any
        ;crosspatch, etc.

        cm1.zusart:
 026023  0 -> mbr                            ;first clear everything
 026024  115.usart + u2651.mcr -> mar(wio)
 026025  do.ios -> upc
 026026  (.+1) -> 11.ret1
 026027  nop                                  ; ... delay to make strobe at least 1
 026030  cm1csr.enall -> mbr                  ;re-enable all aspects
 026031  do.ios -> upc
 026032  (.+1) -> 11.ret1

 026033  u2651.sync.mr1 -> mbr                ;set up for synchronous operation
 026034  115.usart + u2651.mr -> mar(wio)
 026035  do.ios -> upc
 026036  (.+1) -> 11.ret1
 026037  u2651.sync.mr2 -> mbr
 026040  115.usart + u2651.mr -> mar(wio)
 026041  do.ios -> upc
 026042  (.+1) -> 11.ret1

 026043  cm1.syn -> mbr                       ;set up the SYN-1 register
 026044  115.usart + u2651.ssd -> mar(wio)
 026045  do.ios -> upc
 026046  (.+1) -> 11.ret1

 026047  116.crptr -> base
```

```
026050    base -> g1.temp
026051    u2651.sync.cr -> 10, mbr        ;setup and remember the Command regis
026052    g1.temp -> base
026053    115.usart + u2651.cr -> mar(wio)
026054    do.ios -> upc
026055    (.+1) -> l1.ret1

026056    12.ret2 -> upc                  ;return
026057    nop                             ;[1]
```

```
        ;CM1.START, Subroutine, level 2

        ;Call .START with G6 pointing at the DCB and BASE at the right regist
        ;block and it will try to start new IO on the device. This subr is no
        ;called on APR's and PDV's for the device, as well as on IO completio

        ;First, it checks for DCB.IOCB (cached in L4.IOCB)
        ;to be non-zero. If it's non-zero, there's
        ;already an IO going on and we return immediately. If it's zero, we c
        ;IODD.GET in an attempt to get a new IOCB.

        cm1.start:
 026060  l4.iocb ls                    ;is there already an IOCB?
 026061  ifnot zero l2.ret2 -> upc      ;yes, return now

 026062  if zero l0.dcb -> temp         ;set up for IODD.GET call, put DCB in
 026063  iodd.get -> upc                ;no, try to get a new one
 026064  (.+1) -> g6.temp

 026065  if zero l2.ret2 -> upc         ;if none to be had, return
 026066  0 -> l7.next

        ;Here (with dangling UPC change) when there is an IOCB (returned in G
        ;to be output. IO.GET has already setup DCB.IOCB and removed it from
        ;get queue, etc. We now set up the register block to reflect the IOCB
        ;and then return.

 026067  ifnot zero g0.temp -> temp     ;copy the IOCB address into L4.IOCB
 02607C  temp -> l4.iocb

 026071  cm1.restart -> upc             ;"restart" the io which merely sets u
 026072  (.+1) -> l1.ret1               ;the register block from the IOCB

 026073  l2.ret2 -> upc                 ;return
 026074  nop                           ;[1]
```