```
        .TITLE  SBC6120 ROM Monitor


;   Copyright (C) 2001-2003 by R. Armstrong, Milpitas, California.
;   Copyright (C) 1983 by R. Armstrong, Indianapolis, Indiana.
;
;   This program is free software; you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation; either version 2 of the License, or
; (at your option) any later version.
;
;   This program is distributed in the hope that it will be useful, but
; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
; or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
; for more details.
;
;   You should have received a copy of the GNU General Public License along
; with this program; if not, write to the Free Software Foundation, Inc.,
; 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
;


;   This is the ROM monitor for the SBC6120 computer system, which is a Harris
; HM6120 (a PDP-8 for all practical purposes) microprocessor based single board
; computer.  The SBC6120 is intended as a platform for developing other HM6120
; systems and experimenting with various ideas for peripherals, and this gives
; it a rather eclectic mix of hardware:
;
;    * 64KW (that's 64K twelve bit WORDS) of RAM - 32KW for panel memory and
;    32KW for conventional memory.
;
;    * 8KW of EPROM used for bootstrapping - it contains this software.
;
;    * Up to 2Mb (real eight bit bytes this time) of battery backed up,
;    non-volatile SRAM used as a RAM disk for OS/8.  The RAM disk can be
;    mapped into the HM6120 panel memory space.
;
;    * A fairly elaborate memory management system which controls the mapping
;    of RAM, EPROM and RAM disk into panel memory.
;
;    * A real, straight-8, compatible console terminal interface.  The control
;    logic is implemented in a GAL and no 6121 is used (as it is in both
;    models of DECmate).
;
;    * A single digit octal display used to show POST error codes.
;
;    * An IDE disk interface, implemented by a 8255 PPI
;
;   This particular piece of software started life in 1983 as a bootstrap
; program for an elaborate Intersil IM6100 system planned by the author. The
; software was developed and tested on an 6100 emulator running on a
; DECsystem-10, but it never saw any actual hardware.  Although I built several
; simpler 6100 based systems, the one intended for this software proved to be
; too elaborate and complex and was never built.  Until now, that is...


        .HM6120
        .STACK  PAC1, POP1, PPC1, RTN1
        .NOWARN F
        .TITLE  BTS6120 Memory Layout

;   The EPROM in the SBC6120 is only 8K twelve bit words, so the entire code
; for BTS6120 must fit within fields zero and one.  While it's executing,
; however, there is a full 32K of panel RAM available for BTS6120.  Currently
; BTS6120 doesn't use fields two thru seven of panel RAM and these are free.
; They could be used by an OS/8 program via the Copy Memory PR0 function, if
; desired.

; FIELD 0
; Page  Usage
; ----- ----------------------------------------------------------------
; 00000 data & initialized constants
; 00200 SYSINI part 2, later overwritten by the command buffer
```

```
; 00400 command line parsing & error reporting routines
; 00600 RePeat, Terminal, VErsion and HElp commands
; 01000 Examine and Deposit commands
; 01200 examine and deposit subroutines
; 01400 Block Move, ChecKsum, Clear and Fill Memory commands
; 01600 Word Search and Breakpoint List commands
; 02000 Breakpoint set and Remove commands and breakpoint subroutines
; 02200 Proceed, Continue, SIngle step, TRace, Reset ans EXecute commands
; 02400 Boot sniffer, Boot,
; 02600 Partition Map command, and Disk Formatter Pass 1
; 03000 Disk formatter Pass 2, Disk Format, and RAM Disk Format commands
; 03200 BIN (binary paper tape image) loader and LP command
; 03400 Disk (RAM and IDE) Dump and Load commands
; 03600 Disk buffer ASCII dump and load subroutines
; 04000 Partition Copy command
; 04200
; 04400
; 04600
; 05000
; 05200
; 05400
; 05600
; 06000
; 06200 SIXBIT, ASCII, decimal, and octal terminal output
; 06400 address (15 bit) arithmetic, parsing and output
; 06600 keyword scaner and search, decimal and octal input
; 07000 miscellaneous formatted terminal input and output
; 07200 command line scanner
; 07400 terminal input and output primitives
; 07600 control panel entry and context save

; FIELD 1
; Page  Usage
; ----- -----------------------------------------------------------------
; 10000 SYSINI part 1, later overwritten by field 1 variables
; 10200 ROM monitor call (PR0) processor
; 10400 RAM disk R/W, pack/unpack, and battery test routines
; 10600 RAM disk address calculation and diagnostic tests
; 11000 IDE disk initialization and ATA IDENTIFY DEVICE command
; 11200 IDE sector read/write, LBA calculation, wait for READY or DRQ
; 11400 IDE read/write sector buffer, initialize partition map
; 11600 Partition map functions, IDE I/O PR0 call, read/write IDE registers
; 12000 I/O buffer management, Memory Moce PR0 call, cross field calling
; 12200
; 12400
; 12600
; 13000
; 13200
; 13400 to 17377 are used by command tables, error messages and help text
; 17400 to 17777 are used as a temporary buffer for disk I/O
        .TITLE  Edit History


;   1           -- Change documentation of ER and DR commands to reflect the
;                   new format (E <name> and D <name> <value>).
;
;   2           -- Change all the old .SUBTTLs to .TITLE, and remove all .EJECT
;                  pseudo-ops (they are not needed with .TITLE).
;
;   3           -- Invent TDECNW decimal typeout routine.
;
;   4           -- Rename the OUTSTR routine (type an ASCIZ string) to TASCIZ.
;                  Invent the new OUTSTR (type a SIXBIT string), TSIXW, and
;                  TSIXC routines.  Change all messages from ASCII to SIXBIT to
;                  conserve space.
;
;   5           -- Invert the HELLO routine and the VE command to type the
;                  system name and version on startup and command.
;
;   6           -- Remove the SE (set) command and invent the TF, TS and TW
;                  commands to take its place...
;
```

```
;  7           -- Insure that the terminal parameters (width, page size, and
;                 filler class) get initialized to the assembly parameters
;                 defined for that purpose (FTPAGE, FTFILL and FTWIDTH).
;
; 10           -- Make a width value of zero disable the automatic return
;                 feature (so that returns are never inserted).
;
; 11           -- Invent the SM command to set the serial line unit mode
;                 (baud rate, character format, etc).
;
; 12           -- Correct an off by one errors in the BM and CK commands.
;                 Also, make both of these commands check for end of line.
;
; 13           -- Invent the MEMERR routine to type out ?MEM ERR messages.
;                 This is called whenever memory cannot be written properly.
;
; 14           -- Invent the DANDV routine to deposit and verify in main
;                 memory.  This is called by all routines which change memory.
;
; 15           -- Invent new routines to process 15 bit addresses: RDADDR,
;                 NXTADR, TSTADR, and the ?WRAP AROUND error message.
;
; 16           -- Invent the RANGE routine to read address ranges and
;                 change the EXAMINE command to use it (note that this changes
;                 the syntax of examine from $E 0 1 to $E 0<1).
;
; 17           -- Make monitor commands accept trailing blanks at the end of
;                 a line without errors...
;
; 20           -- Revise the deposit command to use 15 bit address routines.
;                 Also make deposit require a ">" character between the
;                 address (or register name) and the data (instead of a
;                 space).
;
; 21           -- Update the BM command to use the new 15 bit addressing.
;                 This now allows transfers to copy more than 4K, and to
;                 cross field boundries.  Also, make it illegal for the source
;                 address to increment out of field 7.
;
; 22           -- Update the CK command for 15 bit addressing.  This makes it
;                 possible to checksum more than 4K of memory in one command.
;
; 23           -- Move the RDMEM routine to page 6400 (the page it was on had
;                 0 words free!).
;
; 24           -- Invent a RAM location KEY and store the WS command search
;                 key there (not in VALUE).  This fixes problems with the WS
;                 command interacting with RDMEM.
;
; 25           -- Make the WS command apply the mask to the search key too.
;                 Also, make WS poll the keyboard so that the operator may
;                 control-C out of a long search.
;
; 26           -- Fix a few bugs in the BM command (these only occur when
;                 transfers cross memory fields). Also change the syntax of
;                 the command to accept a source address range (instead of
;                 a destination address range).
;
; 27           -- Change the name of the SI command to TR (TRace). Then change
;                 the old SN (single instruction with no register output) to
;                 be SI.
;
; 30           -- Re-write the examine register to to save space.
;
; 31           -- Move all the register name strings to the text page.
;                 Re-write the register typeout routines to take less space.
;
; 32           -- Make the examine command accept multiple operands seperated
;                 by commas.
;
; 33           -- Implement the memory diagnostic routines (address test and
;                 data test) and the MD command to execute them.
```

```
;
;   34          -- Invent the DEVERR routine to report device errors.  Also,
;                  re-write MEMERR to share as much code with deverr as
;                  possible.
;
;   35          -- Make the memory data and address diagnostics interruptable
;                  with a control-C (since they take a long time to execute).
;
;   36          -- Invent the REGTST routine to test hardware registers.  This
;                  is used to implement hardware diagnostics.
;
;   37          -- Implement the DPTEST routine to test hardware data paths
;                  for faults.
;
;   40          -- Implement the EX command (to execute IOT instructions).
;
;   41          -- Implement the basic BIN loader routine (BINLOD).
;
;   42          -- Re-arrange code to more tightly pack memory pages (and free
;                  another page of memory).
;
;   43          -- Re-define all IOT instructions to agree with the actual
;                  hardware.
;
;   44          -- Re-write the CP trap and system initialization routines to
;                  work correctly with the latest hardware design (including
;                  the auto-start switches).
;
;   45          -- Add the HALT instruction trap routine (it types out the
;                  message %HALTED @ faaaa)...
;
;   46          -- Ask the user before running diagnostics at startup (even
;                  if he has enabled them via the auto-start switches).
;
;   47          -- Implement the TRAP routine to handle trapped IOTs.
;
;   50          -- Add the code to emulate a KL8E (via trapped IOTs).
;
;   51          -- Fix a bug in the CP trap routine which prevented it from
;                  recognizing SI traps correctly.
;
;   52          -- Make sure the TR command types out the contents of the IR
;                  correctly (add a CLA in an opportune location).
;
;   53          -- Make sure the CONT routine correctly restores the CPU
;                  registers (add another CLA !!!).
;
;   54          -- Revise the monitor command summary in the introduction to
;                  be more up to date.
;
;   55          -- Remove the optional count operand from the TR (trace)
;                  command and always make it execute one instruction instead.
;                  (The user can always combine it with the RP command to get
;                  more than one.)
;
;   56          -- Make the BPT instruction use opcode 6077...
;
;   57          -- The BPT instruction is a trapped instruction, but the
;                  CP entry routine removes breakpoints before the TRAP
;                  routine is called.  Thus TRAP can nenver identify a BPT
;                  instruction. Solution: don't remove breakpoints in the
;                  CP routine, but do it in all the routines that it calls
;                  (except TRAP, of course).
;
;   60          -- Make the BPTINS routine call RDMEM instead of reading memory
;                  itself.
;
;   61          -- Fix the P command so that other commands after it (after
;                  a ';') will work too.  Also, move the P command to the next
;                  memory page (with CONT) to free space.
;
;   62          -- Add the BT command to run the disk bootstrap (which
```

```
;                       presently just types a message to the effect that the disk
;                       bootstrap is not implemented).
;
;  63             -- Implement the following routines: CLRCPU (to clear the
;                       user's CPU registers in RAM), CLRIO (to clear all I/O
;                       devices via a CAF), and CLRBUS (to clear the external
;                       bus by asserting the INITIALIZE signal).
;
;  64             -- Add the MR command to perform a master reset of the CPU and
;                       hardware...
;
;  65             -- Insure that the terminal flag is set after the CLRIO routine
;                       (to avoid hanging the monitor)...
;
;  66             -- Define the PRL instruction (READ2 FTPIEB) to pulse the CPU
;                       RUN/HALT flip-flop.
;
;  67             -- Invent the INIT routine to initialize all hardware which is
;                       important to the monitor (and call it after a hardware reset
;                       command).
;
;  70             -- Make the CONTinue routine clear the console status and set
;                       the RUN flip-flop.
;
;  71             -- Invent the IN (initialize) command to completely initialize
;                       all hardware and software (it is equivalent to pressing the
;                       RESET button !).
;
;  72             -- Be sure the SLU mode gets initialized properly...
;
;  73             -- Modify the CONOUT routine to timeout the console printer
;                       flag and force characters out if it dosen't set after a
;                       reasonable time (this prevents monitor hangs while debugging
;                       programs which clear the flag).
;
;  74             -- Make the hardware diagnostic routine execute a master
;                       reset (i.e. the CLEAR routine) after they are finished
;                       (since they may leave the hardware in a strange state).
;
;  75             -- Change the monitor prompting character to $.
;
;  76             -- Implement the LP command to load paper tapes from the
;                       console...
;
;  77             -- Initialize both the SLU mode and baud rate correctly when
;                       the system is reset...
;
; 100             -- Go through the source and put .ORGs in places where they
;                       have been left out...
;
; After almost 20 years, restart development for the SBC6120 board!
;
; 101             -- A "real" TLS instruction (as the SBC6120 has) doesn't clear
;                       the AC, the way a "fake" TLS (implemented by a 6101 PIE
;                       WRITE) does.  This causes no end of problems for CONOUT and
;                       everything that calls it!
;
; 102             -- Remove the terminal filler code (there's no terminal on the
;                       planet that requires filler characters any more!)
;
; 103             -- Remove the TTY parameters table at TTYTAB.  It's still
;                       possible to set the TTY width and page size with the TW and
;                       TP commands, but they now default to zero (disabled).
;
; 104             -- Remove the place holder code for IOT trapping and KL8
;                       emulation.  It was never used.
;
; 105             -- Expand the command line buffer to occupy all of RAM page 1,
;                       and make the stack always fill whatever remains of page 0.
;
; 106             -- Rewrite to use the HM6120 built in hardware stack.
;
```

```
; 107           -- Some of the old SYSINI code was still left lying around,
;                  and this code would clear RAM on startup.  This is now a
;                  bad idea, since it erases all our page zero vectors!
;
; 110           -- Unlike the 6100 software stack simulation, the 6120 PAC1/2
;                  instructions don't clear the AC.  Add a few CLAs to take
;                  care of this difference.
;
; 111           -- The 6120 stack post decrements after a PUSH, so the stack
;                  pointer needs to be initialized to the top of the stack,
;                  not the TOS + 1...
;
; 112           -- Optimize the page zero constants based on the references
;                  shown in the CREF listing.  Add page zero constants for
;                  popular numerical constants...
;
; 113           -- Do away with the SYNERR link (it's now the same as ZCOMERR).
;
; 114           -- Replace the ERROR link with a simple JMS @ZERROR, and
;                  reclaim the page 0 RAM it used...
;
; 115           -- Do away with RAMINI and the page zero initialization code.
;                  It's no longer necessary, since we now initialize all RAM
;                  from EPROM at startup.
;
; 116           -- Do away with the separate IF, DF and L "registers" and treat
;                  everything as part of the PS (aka flags) register.
;
; 117           -- Move around, clean up, and generally re-organize the
;                  E, D, BM, CK, CM and FM commands.
;
; 120           -- Change the examine register command to ER and the deposit
;                  register command to DR.  Make some other general changes
;                  to the syntax of the E and D commands (e.g. require commas
;                  between multiple items in the deposit list).
;
; 121           -- Change the address range delimiter to "-".
;
; 122           -- Move around, clean up and generally re-organize another
;                  block of code; this time everything concerned with break
;                  points, continue, start, proceed, and CP traps.
;
; 123           -- CONT: needs to use RSP1 to save the monitor's stack pointer,
;                  not LSP1 !!!
;
; 124           -- If a transition occurs on CPREQ L while we're in panel mode,
;                  then the 6120 sets the BTSTRP flag in the panel status
;                  anyway.  This will cause an immediate trap back to panel
;                  mode the instant we try to continue or proceed.  The answer
;                  is to do a dummy PRS before continuing.
;
; 125           -- Move around, clean up and generally re-organize the
;                  remainder of the code.  Move messages and command tables to
;                  field 1.  Change message strings to packed ASCIZ instead of
;                  SIXBIT.
;
; 126           -- The packed ASCIZ OUTSTR routine still has a few non-stack
;                  holdovers - a JMS to OUTCHR and a JMP @OUTSTR to return.
;                  This causes no end of strange symptoms!
;
; 127           -- Use the new \d and \t string escape feature of PALX to put
;                  the system generation time in the monitor.
;
; 130           -- Start adding RAM disk support - add DISKRD and DISKWR
;
; 131           -- Add MCALL (monitor call, via 6120 PR0 trapped IOT)
;
; 132           -- Add DISKRW ROM call function for OS/8 RAM disk driver
;
; 133           -- We're starting to run short of room on page zero, so move
;                  some of the temporary variables that are used only by one
;                  routine to the same page as that routine.  Now that we're
```

```
;                       running out of RAM this is possible.
;
; 134                -- Move the breakpoint tables to page 1, field zero and make
;                       the command buffer that much (about 24 words) smaller.
;                       Since the breakpoint tables are only addressed indirectly
;                       anyway, this hardly made a difference to the code...
;
; 135                -- Fold the MEMERR routine into DANDV and simplify it a little
;                       to save space - just type the good and bad data, and don't
;                       bother with typing the XOR...
;
; 136                -- TOCT3 types the three most significant digits, not the three
;                       least significant !  Oooppppssssss....
;
; 137                -- DDUMP uses COUNT to count the words output, but TOCT4 uses
;                       it to count digits.  Oooppssss...
;
; 140                -- Invent the BSETUP routine and rewrite the break point
;                       functions to use it.  This saves many words of memory.
;
; 141                -- Add the Format Disk (FD) command to "format" RAM disk.  This
;                       is really more of a memory diagnostic than a formatter, but
;                       the name seems appropriate.
;
; 142                -- CALCDA has a off-by-one error that makes it fail for the
;                       last 21 sectors of the RAM disk.
;
; 143                -- Invent the CONFRM routine and use it to make the Format Disk
;                       command ask for confirmation before destroying anything.
;
; 144                -- Add the Disk downLoad command (DL).  This accepts RAM disk
;                       images in exactly the same format as output by DD.
;
; 145                -- Add the H (help) command and a very primitive help system
;                       (it just types out a screen of one line descriptions for
;                       every monitor command!).
;
; 146                -- Add a trace function to the DISKRW MCALL.
;
; 147                -- Accidentally typed a couple of JMSes when I should have
;                       typed .PUSHJ!
;
; 150                -- In DLOAD, we have to use SAVCHR to get the break character
;                       from OCTNW, not GET.
;
; 151                -- IOCLR L (generated by CAF) also clears the console UART,
;                       which causes garbage if we happen to be transmitting a
;                       character at the time.  The easiest way to fix this is to
;                       make the CLEAR routine wait for the console flag to set
;                       (i.e. the transmitter is done) before executing a CAF.
;
; 152                -- The RAM DISK test at TSTUNI doesn't work if the DX pull
;                       down resistors are removed because it fails to mask the
;                       SRAM data to eight bits...
;
; 153                -- Add the infrastructure which allows the code to be split
;                       between field 0 and 1, then move all the low level RAM disk
;                       I/O and ROM call processing code to field 1.
;
; 154                -- Add SP1 and SP2 to the REGLST output (but it's still not
;                       possible to deposit in them or examine them individually).
;
; 155                -- Move the ROM checksums from location 7600 to 0200.
;
; 156                -- Change the hardware so that the TIL311 is loaded via the
;                       6120 WSR (Write to Switch Register) instruction.
;
; 157                -- Begin adding basic IDE/ATA support.
;
; 160                -- Add the GETRDS (Get RAM Disk Size) and GETBAT (get backup
;                       battery status) PR0 functions.
;
```

```
; 161           -- Fix a few bugs in the IDE code, and the basic drive
;                  identification on bootup now works.
;
; 162           -- When programming the 8255 IDE interface, we have to change
;                  the mode (input vs output) _before_ we set up the address
;                  bits.  If we don't then the address bits get cleared by
;                  the #$&(*# 8255 when the mode is changed!
;
; 163           -- Add Get and Set Disk Partition mapping PR0 subfunctions.
;
; 164           -- Add Get IDE Disk Size subfunction to PR0.
;
; 165           -- Add the Copy Memory subfunction to PR0.
;
; 166           -- Add the last missing bits of IDE disk I/O.  We're now ready
;                  to try out the OS/8 device handler.
;
; 170           -- INIPMP needs to do a CLL to ensure that the link is in a
;                  known state - otherwise it can sometimes quit too soon!
;
; 171           -- Extend the BOOT command to boot either IDE or RAM disk.
;                  Add the "boot sniffer" to determine whether a volume
;                  contains a real bootstrap and use it to make "B" with no
;                  arguments search for a bootable volume (just like a real
;                  computer!).
;
; 172           -- The disk download routine needs to call PNLBUF before it
;                  attempts to write the buffer...
;
; 173           -- Make illegal PR0 functions print a message and return to
;                  BTS6120 command level.  This solves the problem of how to
;                  skip over the arguments for a call we don't understand.
;
; 174           -- Make the MR (master reset) command initialize the IDE
;                  drive and reset the partition map.
;
; 175           -- Add the PM command to edit/view the disk partition map.
;
; 176           -- Completely rewrite the RAM disk formatter code to create
;                  two commands, RF to format a RAM disk an DF to format an
;                  IDE disk partition.
;
; 177           -- PMEDIT screws up the disk LUN - there's a DCA that should
;                  have been a TAD!
;
; 200           -- Fix a few bugs in the DF and RF routines - FMTARG is missing
;                  a .POPJ, and the unit number gets corrupted in FMTARG
;                  by the TOCT4S routine, which uses WORD.
;
; 201           -- The pin that corresponds to A17 on a 512K chip is an
;                  alternate chip select on the 128K chips.  Worse, this extra
;                  chip select is active HIGH, which means that A17 must always
;                  be set before the 128K chips will do anything!
;
; 202           -- Clean up all the help messages and make sure they agree
;                  with the current state of all the commands.
;
; 203           -- Do away with F1CALL and create PUSHJ1 instead.  Both allow
;                  field 0 routines to call field 1, however the new one takes
;                  one less word of code at the point of the call, which
;                  helps a lot on some pages where words are tight.
;
; 204           -- Invent a few more page zero constants and use them to help
;                  out on a few more pages where space is tight.
;
; 205           -- Make the disk related commands (DL, DD, and DF) verify that
;                  an IDE disk is really present by checking for DKSIZE != 0.
;                  Also make the DISKRW PR0 call return error -1 if no disk
;                  is attached.
;
; [Start porting to the SBC6120 model 2 hardware]
;
```

```
; 206           -- Remove all the bicolor LED stuff - it doesn't exist in
;                  the SBC6120 model 2.
;
; 207           -- The model 2 schematic accidentally switched CS1FX/CS3FX
;                  and DIOR/DIOW in the IDE interface.  Modify the code to
;                  paramaterize all IDE interface bits and then chage these
;                  parameters to agree with the real hardware.
;
; 210           -- Remove all the unnecessary CLAs after PPI instructions
;                  (in the model 2, these IOTs all clear the AC).
;
; 211           -- Fix the timeouts for the console flag and the LP command
;                  so they're reasonable given the 4.9152Mhz clock of the
;                  SBC6120 model 2.
;
; [SBC6120 now runs on the Model 2 hardware!]
;
; 212           -- Fix a nasty bug in the LBA calculation!
;
; 213           -- Fix TDECNW to handle unsigned numbers up to 4095
;
; 214           -- Change the "RAM: " in the RAM disk message to "NVR:"
;                  to avoid confusion with the main memory..
;
; 215           -- Add the PC (partition copy) command.
;
; [End of monitor edit history]
VERSION=215     ; latest edit number
        .TITLE  SBC6120 IOTs and Definitions


;   The console terminal interface of the SBC6120 is actually straight -8
; compatible, which is a proper subset of the KL8E except that the KCF, TFL,
; KIE and TSK (or SPI, depending on which manual you read!) instructions are
; omitted.  Console interrupts are permanently enabled, as they were in the
; original PDP-8.  The console interface in the SBC6120 DOES NOT use a 6121,
; so there'll be none of this skip-on-flag-and-clear-it nonsense with KSF or
; TSF!
KSF=6031        ; Skip of console receive flag is set
KCC=6032        ; Clear receive flag and AC
KRS=6034        ; OR AC with receive buffer and DON't clear the flag
KRB=6036        ; Read receive buffer into AC and clear the flag
TSF=6041        ; Skip if the console transmit flag is set
TCF=6042        ; Clear transmit flag, but not the AC
TPC=6044        ; Load AC into transmit buffer, but don't clear flag
TLS=6046        ; Load AC into transmit buffer and clear the flag

; 8255 PPI Interface IOTs...
PRPA=6470       ; read PPI port A
PRPB=6471       ;  "    "   "   B
PRPC=6472       ;  "    "   "   C
PRCR=6473       ;  "    "  control register
PWPA=6474       ; write PPI port A
PWPB=6475       ;   "    "    "  B
PWPC=6476       ;   "    "    "  C
PWCR=6477       ;   "    "  control register

; Other SBC6120 instructions...
POST=6440       ; Display a 4 bit POST code
BPT=PR3         ; Breakpoint trap instruction

; Special ASCII control characters that get used here and there...
CHNUL=000       ; A null character (for fillers)
CHCTC=003       ; Control-C (Abort command)
CHBEL=007       ; Control-G (BELL)
CHBSP=010       ; Control-H (Backspace)
CHTAB=011       ; Control-I (TAB)
CHLFD=012       ; Control-J (Line feed)
CHCRT=015       ; Control-M (carriage return)
CHCTO=017       ; Control-O (Suppress output)
CHXON=021       ; Control-Q (XON)
CHCTR=022       ; Control-R (Retype command line)
```

```
CHXOF=023         ; Control-S (XOFF)
CHCTU=025         ; Control-U (Delete command line)
CHESC=033         ; Control-[ (Escape)
CHDEL=177         ; RUBOUT    (Delete)

; OPR microinstructions that load the AC with various special constants...
NL0000=CLA                        ; all models
NL0001=CLA IAC                    ; all models
NL0002=CLA CLL CML     RTL        ; all models
NL2000=CLA CLL CML     RTR        ; all models
NL3777=CLA CLL     CMA RAR        ; all models
NL4000=CLA CLL CML     RAR        ; all models
NL5777=CLA CLL     CMA RTR        ; all models
NL7775=CLA CLL     CMA RTL        ; all models
NLM3=NL7775                       ; all models
NL7776=CLA CLL     CMA RAL        ; all models
NLM2=NL7776                       ; all models
NL7777=CLA         CMA            ; all models
NLM1=NL7777                       ; all models
NL0003=CLA STL IAC RAL            ; PDP-8/I and later
NL0004=CLA CLL IAC RTL            ; PDP-8/I and later
NL0006=CLA STL IAC RTL            ; PDP-8/I and later
NL6000=CLA STL IAC RTR            ; PDP-8/I and later
NL0100=CLA IAC BSW                ; PDP-8/E and later
NL0010=CLA IAC R3L                ; HM6120 only
        .TITLE  SBC6120 Memory Mapping Hardware


;   The SBC6120 has three memory subsystems - 64K words of twelve bit RAM,
; 8K words of 12 bit EPROM (actually the EPROM is 16 bits wide, but the
; hardware just throws away the extra four bits), and up to 2Mb of 8 bit
; battery backed up SRAM for a RAM disk.
;
;   The HM6120 on the other hand, has only two memory spaces - panel memory
; and main memory, and each of these is limited to 32K words.  The SBC6120
; implements a simple memory mapping scheme to allow all three memory
; subsystems to fit in the available address space.  Up to four different
; memory maps are possible, although only three are currently implemented.
;
;   The memory map in use is selected by four IOT instructions, MM0, MM1
; MM2 and (what else) MM3.  Memory map changes take place immediately with
; the next instruction fetch - there's no delay until the next indirect JMP
; the way there is with a CIF instruction.
;
;   The four memory maps implemented by the SBC6120 are:
;
; * Map 0 uses the EPROM for all direct memory accesses, including instruction
;   fetch, and uses the RAM for all indirect memory accesses.  This is the
;   mapping mode set by the hardware after a power on reset.
;
; * Map 1 uses the RAM for all direct memory accesses, including instruction
;   fetch, and uses the EPROM for all indirect memory references.  This mode
;   is the "complement" of map 0, and it's used by the panel memory bootstrap
;   to copy the EPROM contents to RAM.
;
; * Map 2 uses the RAM for all memory accesses, regardless.  This is the
;   mapping mode used during almost all operation after booting.
;
; * Map 3 is the same as map 2, except that the RAM disk memory is enabled
;   for all indirect accesses.  BEWARE - RAM disk memory is only eight bits
;   wide and reads and writes to this memory space only store and return the
;   lower byte of a twelve bit word.  This mode is used only while we're
;   accessing the RAM disk.
;
;   IMPORTANT!  The memory mapping mode affects only 6120 control panel memory
; accesses.  Main memory is always mapped to RAM regardless of the mapping
; mode selected.

                  ;   DIRECT      INDIRECT
                  ;   --------    --------
MM0=6400          ;   EPROM       RAM      (automatically selected by a RESET)
MM1=6401          ;   RAM         EPROM    (used during system initialization)
```

```
MM2=6402        ;   RAM         RAM     (used almost all the time)
MM3=6403        ;   RAM         DISK    (used to access RAM disk only)
        .TITLE  System Startup and POST Codes


;   Getting the SBC6120 monitor up and running after a power up is a little
; harder than we might wish.  Our first problem is that we're actually
; executing code from the EPROM now, and a lot of the usual PDP-8 techniques
; of "self manipulation" (e.g. a JMS instruction!) won't work because the
; program store isn't writable.  Our second problem is that all of panel
; fields 0 and 1 are mapped to EPROM, and there's no RAM anywhere in these
; fields at all, not even in page zero.
;
;   Our final problem is that we can't even be sure that all the hardware is
; working correctly at this point.  If some part isn't working, for example,
; the RAM, we'd like to provide at least some kind of diagnostic message
; before we end up lost forever.  The minimum set of system components that
; this monitor needs to have working before it can print a prompt and execute
; commands is 1) the CPU, 2) the ROM, 3) the RAM, and 4) the console terminal.
;
;   This means we need to accomplish two things during a cold boot - first,
; to execute a simple power on self test (aka POST), and second, to copy this
; monitor's code from EPROM to panel RAM where we can execute it without
; restriction.
;
;   The SBC6120 has a single digit LED display that the program can set, via
; the POST instruction.  At the beginning of each step in testing and
; initialization we set this digit to a particular value, and then if that
; test or startup part fails, we simply halt and the display remains at the
; last value set.  The digits and their associated failure modes are:
;
;       7 - CPU failure (or it's not a 6120)
;       6 - panel RAM bootstrap failure
;       5 - RAM checksum failure
;       4 - memory test failure
;       3 - currently unused (reserved for 6121 failure?)
;       2 - console terminal failure
;       1 - panel monitor running (success!)
;       0 - user (main memory) program running
        .TITLE  System Startup, Part 1


;   This code lives in field one, page zero of the EPROM and it's the first
; thing that gets executed after a power on clear.   Its main job is to get
; the SBC6120 memory initialized, which it does it with these steps:
;
;       1 - execute a simple CPU test
;       2 - execute a very simple RAM test
;       3 - copy EPROM to panel RAM
;       4 - verify the firmware checksum
;       5 - execute an extensive test on the remaining memory
;
;   After it's done, it jumps to the second phase of initialization, which
; lives in field zero.  When this code starts we're in memory mapping mode
; zero, which means that all instructions are being executed from EPROM and
; RAM can be addressed only indirectly.  When it finishes, all code will be
; in panel RAM and we'll be running in memory mapping mode two, which means
; that all memory accesses go to RAM and the EPROM is inaccesible.
;
;   After system initialization is complete, all this code is over written
; with page zero variables and storage for field one.
        .FIELD  1
        .PAGE   0

;   Location zero of field 1 (and field 0, for that matter) must contain a
; zero for the checksum routine.  See the code at ROMCHK: for a discussion.
        0000

;   The first step is the CPU test, which is trivially simple.  We just
; verify that we're actually running on a HM6120 and let it go at that...
SYSINI: POST+7                  ; set the POST code to 7
        CLA IAC R3L             ; Only a 6120 has the R3L instruction
```

```
        TAD     [-10]           ; Did we get the right answer?
        SZA                     ; ???
         JMP     .              ; Nope - halt forever

;   Before we copy the boot code to panel RAM, do a primitive (and it's really
; primitive!) test of RAM just to make sure there's something there we can
; read and write.  Remember that at this point we're using memory map 0, so
; all direct references are to EPROM, but all indirect references are to RAM.
        POST+6                  ; set the POST code to six
        SPD                     ; be sure we're accessing panel memory now!
        CLA                     ; ...
        TAD     [2525]          ; write an alternating bit pattern
        DCA     @[ROMCPY]       ;  ... to RAM location zero
        TAD     [5252]          ; and write the complement to location 1
        DCA     @[ROMCPY+1]     ;  ...
        TAD     @[ROMCPY]       ; now read them both back
        TAD     @[ROMCPY+1]     ; and add them up
        CMA                     ; and the result should be -1
        SZA                     ; ????
         JMP     .              ; low RAM failure

;   Copy all of the EPROM moving code, which is six words starting at the label
; ROMCPY, from EPROM to exactly the same location in RAM.  There's no way to
; use a loop to do this since we don't have any RAM that we can access directly
; to use as a pointer!
        TAD     ROMCPY          ; copy this one word from EPROM
        DCA     @[ROMCPY]       ;  ... to RAM
        TAD     ROMCPY+1        ; and do it for the entire routine
        DCA     @[ROMCPY+1]     ; ...
        TAD     ROMCPY+2        ; ...
        DCA     @[ROMCPY+2]     ; ...
        TAD     ROMCPY+3        ; ...
        DCA     @[ROMCPY+3]     ; ...
        TAD     ROMCPY+4        ; ...
        DCA     @[ROMCPY+4]     ; ...
        TAD     ROMCPY+5        ; ...
        DCA     @[ROMCPY+5]     ; ...

;   Now it gets tricky.  At this instant we're still running in EPROM, and the
; first two instructions (at ROMCP0) get executed from EPROM.  As soon as we
; execute the MM1 at ROMCPY:, however, the very next instruction is fetched
; from RAM.  This should work, because the previous code has copied the six
; words that make up the ROMCPY loop from EPROM to exactly the same location
; in RAM.
;
;   The loop the copies an entire field from EPROM to RAM, executing from RAM
; the whole time.  It actually over writes itself in the process, but since it
; over writes itself with a copy of the exact same code we should be OK.  By
; the time it falls thru the ISZ at the end of the loop, the subsequent code
; should exist in RAM.
;
;   After copying field 1, we switch to field zero and jump back to ROMCP0
; to do it again.  Although the first time thru we executed ROMCP0 from EPROM,
; the second time thru we execute it from RAM, which is OK because it got
; copied during the first pass.
;
;  Say goodbye to memory map 0 - we'll never need it again!

;   This loop copies all of a field, except location 0, from EPROM to RAM.
ROMCP0: CLA IAC                 ; always start with location 1, not zero
        DCA     @[0]            ; save the address pointer here
ROMCPY: MM1                     ; (1) address RAM directly, EPROM indirectly
        TAD     @0              ; (2) and load a word from EPROM
        MM2                     ; (3) address RAM for all memory accesses
        DCA     @0              ; (4) and store the word in RAM
        ISZ     0               ; (5) have we done an entire field?
         JMP     ROMCPY         ; (6) nope - keep copying
        RDF                     ; which field did we just copy?
        CDF     0               ; assume that we'll copy field zero next
        SZA CLA                 ; but did we just copy field zero?
         JMP     ROMCP0         ; no - go copy it
```

```
;   When we leave this loop, we're using memory map 2 which means panel RAM
; is used everywhere and the EPROM is inaccessible.  We'll stay in this mapping
; mode forever, except when we're accessing the RAM disk.

;  Each field of the panel ROM contains a 12 bit checksum, put there by the
; PDP2HEX program and calculated so that the sum of all words in the field,
; including the checksum word, will be zero.  Now we'll compute and verify the
; checksum of both RAM fields, which proves that our RAMs work, that the
; EPROMS were programmed correctly, and that we copied them correctly.
;
;   It might seem a better to do this before on the EPROMs before we've copied
; them to RAM, but the answer is that it's just impossible to compute a
; checksum using memory map 0 - there's no directly addressible RAM to use for
; address pointers or for storing the sum!
;
;   One last subtle point is that we're keeping an address pointer in location
; zero of RAM, which wasn't in the EPROM when PDP2HEX calculated its checksum.
; This actually works by accident (er - excuse me, "Design"), since we keep
; our pointer in location zero of RAM, it will have the value zero when we
; checksum it.  Coincidentally, this is exactly the same value that's in
; location zero of the ROM image.

; This loop checksums one RAM field...
        POST+5                  ; set the POST code to five
        CDF     1               ; checksum field 1 first
ROMCHK: CLA IAC                 ; and start with location 1 (skip zero)
        DCA     0               ; ...
ROMCH0: TAD     @0              ; add up another word from RAM
        ISZ     0               ; have we done an entire field?
         JMP     ROMCH0         ; nope - keep adding
        SZA                     ; yes - did the checksum pass?
         JMP     .              ; RAM checksum failure
        RDF                     ; get the field we just did
        CDF     0               ; and assume we'll do field zero next
        SZA CLA                 ; but did we really just do zero?
         JMP     ROMCHK         ; no - go checksum it now

;   The next step is to run a memory test on the remaining fields (2 thru 7)
; of panel memory and all fields of main memory.  It's not a very sophisticated
; test - it just writes each memory location with its address in the first pass
; and then reads it back in the second, but it does prove that there's at least
; some memory out there listening.
;
;   Before we do that, however, we do an even simpler test to verify that the
; panel data flag is working and that main memory and panel memory are two
; separate and distinct memory spaces...

; Make sure that panel memory and main memory are distinct...
        POST+4                  ; set the POST code to four
        STA                     ; put -1 in location 0 of panel memory
        DCA     @[0]            ; ...
        CPD                     ; and then put 0 in the same location
        DCA     @[0]            ;  ... of main memory
        SPD                     ; back to panel memory
        ISZ     @[0]            ; and increment -1
         JMP     .              ; if it doesn't skip something is wrong

; Test all eight fields of main memory...
;       CLA                     ; and start testing with field zero
        CPD                     ; address main memory again
MEMTS1: JMS     FLDTST          ; test this field, halt if it's bad
        SZA                     ; have we wrapped around to field 0 again ?
         JMP     MEMTS1         ; no - test the next field

; Then test only fields 2 thru 7 of panel memory...
        SPD                     ; address panel memory this time
        TAD     [20]            ; start testing with field 2
MEMTS2: JMS     FLDTST          ; test this field, halt if it's bad
        SZA                     ; ...
         JMP     MEMTS2         ; ...

;   System initialization, part 1, is finished.  The remainder of the code
```

```
; lives in field zero...
        CXF     0
        JMP     @[SYSIN2]

;   This subroutine will test one field of either main memory or panel memory.
; It's a fairly simple minded test - it just writes each location with its
; address in the first pass and then reads it back in the second pass.  If the
; test fails it just halts - there is no error return!
FLDTST: 0                       ; enter here with the field in the AC
        TAD     [CDF 0]         ; make a CDF instruction out of it
        DCA     .+1             ; and execute it inline
         NOP                    ; gets over written with a CDF
        DCA     0               ; reset the address pointer
FLDTS1: TAD     0               ; write each word with its address
        DCA     @0              ; ...
        ISZ     0               ; have we done all 4K?
         JMP     FLDTS1         ; nope - keep going
        DCA     0               ; yes - reset the address pointer
FLDTS2: TAD     0               ; and make another pass to test
        CIA                     ;  ... what we wrote
        TAD     @0              ; ...
        SZA                     ; does this location contain the right value?
         JMP     .              ; no - just halt
        ISZ     0               ; yes - keep going for all 4K
         JMP     FLDTS2         ; ...
        RDF                     ; get the data field we just tested
        TAD     [10]            ; and increment it for the caller
        AND     [70]            ; remove any overflow bits
        JMP     @FLDTST         ; return the next field to the caller

        .PAGE
        .TITLE  System Startup, Part 2


;   This routine is the second phase of system initialization, and it lives
; in page one of field zero.  It's called at the end of the first phase, and
; it will:
;
;       1 - test and initialize any extra hardware (e.g. 6121 chips)
;       2 - test and initialize the console terminal
;       3 - initialize any RAM that needs it
;       4 - print a sign on message
;       5 - jump to the monitor restart address
;
;  After this code finishes, the monitor is running and a prompt will have been
; printed on the terminal.  This code code gets overwritten immediately by the
; monitor's command line buffer, which also lives in page 1 of field 0.
        .FIELD  0
        .PAGE   1

;  The PDP2HEX program (which converts BIN files into ROM images in Intel
; HEX format) stores a checksum of ROM field 0 in location 00200, which will
; later be used by the POST...
ROMCK0: .BLOCK  1

;   Some space is reserved here for initializing the hardware, especially any
; 6121 chips that might be lying around.  We don't currently have any of those,
; so we don't worry about it now.
SYSIN2: POST+3                  ; set the POST code to three

;  The next stage in initialization is to test the console terminal.  Since
; the SBC6120 hardware doesn't have a loopback mode we can't really verify that
; data is being sent and received correctly, but we can at least test that the
; flags set and clear at appropriate times. That way we'll know, at last, that
; we won't hang forever if we do a "TLS; JMP .-1" loop.
        POST+2                  ; set the POST code to two
        CLA                     ; send a null character to the console
        TLS                     ; ...
        TSF                     ; and then wait for the flag to set
         JMP     .-1            ; waiting forever, if necessary!
        TCF                     ; clear the console flag
        TSF                     ; and then test it again
```

```
        SKP                     ; good - it _isn't_ set!
        JMP     .               ; bad - it's still set, so the console fails
       TLS                      ; send another null
       TSF                      ; and be sure it sets one more time
        JMP     .-1             ; ...

;  Now make sure we can clear the keyboard input flag, and that KCC also
; clears the AC. The latter proves that there is at least some hardware out
; there controlling the C lines for the console terminal, although it doesn't
; guarantee that we can receive data.
        STA                     ; Load the AC with -1
        KCC                     ; Clear the keyboard flag and the AC
        SZA                     ; Verify that the AC got cleared
         JMP    .               ; Nope - console test failed!
        KSF                     ; And test the keyboard flag
         SKP                    ; Good - it _isn't_ set!
         JMP    .               ; Bad - the keyboard test failed

; Print a sign on message.
SYSIN3: POST+1                  ; the monitor is up and running now


;   This code starts up the monitor/bootstrap after a system reset.  It
; initializes the monitor RAM, sets up the stack, and jumps to the monitor
; entry point.  Since we don't know how we came to be here, this code
; shouldn't make any assumptions about the current state of the hardware!
        CXF     0               ; Be sure the IB and DF are both zero
        SPD                     ; Address CP memory with indirect cycles
        CLA                     ; just in case...
        TAD     [STACK]         ; reset the 6120 stack pointer
        LSP1                    ; ...

;  Set the control panel entry vector in 7777 to be a "JMP CPSAVE" instruction.
; We have to do this in a rather awkward way because PALX won't assemble a
; current page reference to CPSAVE unless we're actually on the same page as
; CPSAVE!
        TAD     [CPSAVE&177 | 5200]
        DCA     @[7777]

; Do any RAM initialization that needs to be done...
        TAD     [80.]           ; the default terminal width is 80
        DCA     WIDTH           ; ...
        DCA     LENGTH          ; and automatic XOFF is disabled
        .PUSHJ  @[CLRCPU]       ; clear the saved user context
        .PUSHJ  @[BPTCLR]       ; clear the breakpoint tables
        JMS     @ZPUSHJ1        ; (cross field call)
         INIPMP                 ; initialize the IDE disk partition map

; Type out the system name...
        .PUSHJ  @ZCRLF          ; First start on a new line
        .PUSHJ  @[HELLO]        ; Finally add our name and version

;   Now we are ready to initialize the RAM disk array by first testing the
; backup battery and then individually testing each of the four RAM chips to
; determine a) if one is installed, and b) how big it is.  IMPORTANT - because
; of the way the DS1321 works, we MUST test the backup battery before any
; other accesses to the RAM disk!  The RDTEST routine will automatically
; initialize the RDSIZE array with the size of each RAM disk chip that it
; discovers...
        JMS     @ZINLMES        ; say
         RAMMS1                 ; "RAM disk: "
        JMS     @ZPUSHJ1        ; (cross field call)
         BATTST                 ; test the backup battery state
        JMS     @ZPUSHJ1        ; (cross field call)
         RDTEST                 ; test all four RAM disk units
        .PUSHJ  @[TDECNW]       ; type out the total RAM size
        JMS     @ZINLMES        ; say
         RAMMS3                 ; "Kb - Battery "
        CDF     1               ; the battery OK flag lives in field 1
        TAD     @[BATTOK]       ; get the battery status flag
        CDF     0               ; ...
        SNA CLA                 ; is the batery OK?
```

```
         TAD    [BFAMSG-BOKMSG] ; no - say "failed"
        TAD     [BOKMSG]        ; yes - say "OK"
        .PUSHJ  @[OUTSTR]       ; ...
        .PUSHJ  @ZCRLF          ; finish the status report and we're done

;   Finally, probe the IDE bus for any drive that might be attached.  First
; we have to initialize the 8255 and reset the IDE bus, and then we can send
; an ATA IDENTIFY DEVICE command to the drive.  The DISKID routine will
; extract the drive's capacity, in MB, from that and leave the result at
; DKSIZE.  DISKID also leaves the first 256 bytes of the drive's response in
; the DSKBUF, and we can use that to type out the drive's make and model,
; which appears there in plain ASCII.
        JMS     @ZINLMES        ; say
         IDEMS1                 ; "IDE disk: "
        JMS     @ZPUSHJ1        ; (cross field call)
         IDEINI                 ; initialize the IDE interface
        SZL                     ; is there a drive attached?
         JMP     SYSIN4         ; nope - quit now
        JMS     @ZPUSHJ1        ; (cross field call)
         DISKID                 ; send an IDENTIFY DEVICE command to the drive
        SZL CLA                 ; did it work ?
         JMP     SYSIN4         ; nope - there's no disk there after all
        CDF     1               ; disk data lives in field 1
        DCA     @[DSKBUF+135]   ; (make the model string ASCIZ)
        TAD     @[DKSIZE]       ; get the total disk size
        CDF     0               ; ...
        SNA                     ; is the disk size zero ?
         JMP     SYSIN7         ; yes - this disk is "unsupported" !
        .PUSHJ  @[TDECNW]       ; and type it out in decimal
        JMS     @ZINLMES        ; say
         IDEMS2                 ;  "MB"
        TAD     [DSKBUF+66-1]   ; point to the make/model string
        .PUSHJ  @[TASZF1]       ; and type that out, in ASCII
        JMP     SYSIN5          ; go type a CRLF and we're done

;   Here if an unsupported (i.e. one which does not support LBA addressing)
; is detected...
SYSIN7: JMS     @ZINLMES        ; say
         IDEMS4                 ; "not supported"
        JMP     SYSIN5          ; ...

; Here if no IDE disk is detected...
SYSIN4: JMS     @ZINLMES        ; and say
         IDEMS3                 ;  "NONE"
SYSIN5: .PUSHJ  @ZCRLF          ; finish the line and we're done

; And we're ready for commands...
         JMP     @ZRESTA

        .PAGE
        .TITLE  Field 0 Variables


;   Page zero of field zero contains most of the runtime data for the monitor,
; including the saved state of the user (main memory) program.  It is purposely
; NOT overlayed by any startup code so that it may also contain initialized
; variables, such as JMP/JMS vectors.  Data in this page gets initialized
; automatically when the first phase of system initialization copies the EPROM
; to RAM.

;   These words contain the saved state of the main memory (aka user)
; program.  The PC gets saved to location zero automatically by the 6120
; on any entry to control panel, and the rest get saved by the code around
; CPSAVE...
        .ORG    0000
UPC:    .BLOCK  1       ; program counter (saved by the hardware)
UAC:    .BLOCK  1       ; accumulator
UFLAGS: .BLOCK  1       ; status (LINK, GT, IF, DF, etc) from GCF
UMQ:    .BLOCK  1       ; MQ register
USP1:   .BLOCK  1       ; 6120 stack pointer #1
USP2:   .BLOCK  1       ;   "    "      "    #2
UIR:    .BLOCK  1       ; the last main memory instruction to be executed
```

```
; Auto-index registers...
        .ORG    0010    ; this must be at location 10 !!!
X1:     .BLOCK  1       ; the first auto-index register
X2:     .BLOCK  1       ; the second auto-index register
X3:     .BLOCK  1       ; the third auto-index register
L:      .BLOCK  1       ; the command line pointer
        .ORG    0020    ; don't put anything else in auto-index locations

; Command parameters...
ADDR:   .BLOCK  1       ; the current address
ADRFLD: .BLOCK  1       ; the field of this command
PNLMEM: .BLOCK  1       ; non-zero if ADDR/ADRFLD references panel memory
HIGH:   .BLOCK  1       ; the high end of an address range
LOW:    .BLOCK  1       ; the low end of an address range
HGHFLD: .BLOCK  1       ; the field of the high address
LOWFLD: .BLOCK  1       ; the field of the low address
VALUE:  .BLOCK  1       ; the data word or value
NAME:   .BLOCK  1       ; the name of this command
CHKSUM: .BLOCK  1       ; the checksum of memory or tape
SIMFLG: .BLOCK  1       ; non-zero if we're executing a single instruction
SAVCHR: .BLOCK  1       ; a place to save a character

; Terminal parameters...
CTRLO:  .BLOCK  1       ; non-zero if output is supressed
XOFF:   .BLOCK  1       ; non-zero if output is suspended
HPOS:   .BLOCK  1       ; the current horizontal position
VPOS:   .BLOCK  1       ; the current vertical position
WIDTH:  .BLOCK  1       ; the width of the terminal
LENGTH: .BLOCK  1       ; the number of lines on the screen
IRMA:   .BLOCK  1       ; the console flag timeout counter

; Parameters for the repeat command...
REPCNT: .BLOCK  1       ; the number of times to repeat
REPLOC: .BLOCK  1       ; the location to repeat from

; Number I/O locations...
WORD:   .BLOCK  1       ; the number being read or written
WORDH:  .BLOCK  1       ; the high order bits of the last word read
COUNT:  .BLOCK  1       ; the number of digits read or written
DIGITS: .BLOCK  1       ; counts digits for numeric input/output routines

; Storage for RDDUMP, DDDUMP, RLLOAD and DLLOAD, RFRMAT, and DFRMAT...
RECSIZ: .BLOCK  1       ; disk (page, block) size
RECCNT: .BLOCK  1       ; number of records to dump
FMTCNT: .BLOCK  1       ; number of blocks/records processed by FORMAT
CPYSRC=HIGH             ; source partition for PC command
CPYDST=LOW              ; destination partition for PC command

; Page zero vectors (to save literal space)...
ZOUTCHR:  OUTCHR        ; type a single character
ZTSPACE:  TSPACE        ; type a space
ZTOCT4:   TOCT4         ; type an octal number
ZTOCT4C:  TOCT4C        ; type an octal number followed by a CRLF
ZTOCT4S:  TOCT4S        ; type an octal number followed by a space
ZCRLF:    CRLF          ; type a carriage return/line feed
ZINLMES:  INLMES        ; type a string passed in-line
ZSPACMP:  SPACMP        ; get the next non-blank command character
ZSPACM0:  SPACM0        ; get a non-blank character starting with the current
ZBACKUP:  BACKUP        ; backup the command line pointer
ZEOLTST:  EOLTST        ; test current character for end of line
ZEOLNXT:  EOLNXT        ; test the next character for end of line
ZGET:     GET           ; get the next character from the command line
ZOCTNW:   OCTNW         ; scan an octal number
ZRANGE:   RANGE         ; scan an address range (e.g. "0-7777")
ZTSTADR:  TSTADR        ; compare the HIGH/HGHFLD to LOW/LOWFLD
ZNXTADR:  NXTADR        ; increment ADDR/ADRFLD
ZRDMEM:   RDMEM         ; read a word from main or panel memory
ZDANDV:   DANDV         ; deposit (in memory) and verify
ZRESTA:   RESTA         ; monitor restart vector
ZCOMERR:  COMERR        ; report a command syntax error and restart
ZERROR:   ERROR         ; print an error message and restart
```

```
ZPUSHJ1:  PUSHJ1        ; call a routine in field 1 and return to field 0

; Page zero constants (to save literal space)...
ZK177:     177          ; used everywhere as a mask for ASCII characters
ZK70:       70          ; used as a mask for data/instruction fields
ZK7:         7          ; yet another mask
ZMSPACE:  -" "          ; an ASCII space character (or the negative there of)
ZM128:    -128.         ; record size of RAM disk
ZK7600=ZM128
ZM256:    -256.         ; record size of IDE disk
ZK7400=ZM256
ZRDPAGE: RDPAGE         ; current RAM disk page number in field 1
ZDKRBN:  DKRBN          ; current IDE disk block number in field 1

; The software stack occupies all of the rest of page zero.
STKSAV: .BLOCK  1       ; the last monitor SP is saved here by CONTINUE
STACK=0177
STKLEN=STACK-.          ; Length of the stack (if anybody cares)


;   Page one of field zero contains the second phase system initialization
; code, and it's over written by the command line buffer and break point
; tables after we're running.
        .ORG    0200

;  The PDP2HEX program stores a checksum of ROM field 0 in location 00200,
; and we have to reserve space for it here so it doesn't get overwritten by
; any of our data.  See the code at ROMCK0: for more discussion.
        .BLOCK  1

; Breakpoint storage...
MAXBPT=8.               ; maximum number of breakpoints
BPTADR: .BLOCK  MAXBPT  ; address assigned to each breakpoint
BPTFLD: .BLOCK  MAXBPT  ; field of the breakpoint
BPTDAT: .BLOCK  MAXBPT  ; original data at the breakpoint
BPTEND=BPTADR+MAXBPT-1  ; end of the breakpoint address table

; The command line buffer for INCHWL occupies all that remains of page one...
MAXCMD=0400-.           ; space available for the command buffer
CMDBUF: .BLOCK  MAXCMD  ; and the actual command buffer
        .TITLE  Monitor Main Loop

        .PAGE   2

;   This routine will read commands from the terminal and execute them.  It
; can be entered at RESTA to restart after a control-C or a fatal error, and
; at BOOTS after completion of a normal command...
RESTA:  SPD                     ; Insure that CP memory is always selected
        CLA                     ; And be sure the AC is cleared
        TAD     [STACK]         ; Point to the stack
        LSP1                    ; Clean up the stack pointer
        .PUSHJ  @ZCRLF          ; Be sure the terminal is ready

; Read another command line...
BOOTS:  CLA                     ; ...
        TAD     [">"]           ; Point to the prompt
        .PUSHJ  @[INCHWL]       ; And read a command line
        DCA     REPCNT          ; Clear the repeat counter initially

; Execute the next command...
BOOTS1: .PUSHJ  @[NAMENW]       ; First identify a command
        TAD     NAME            ; Get the name we read
        SNA CLA                 ; Is this a null command ??
         JMP     BOOTS2         ; Yes -- just ignore it
        TAD     [CMDTBL-1]      ; Then point to the list of commands
        .PUSHJ  @[MATCH]        ; And look it up

; See if there are more commands on this line...
BOOTS2: TAD     L               ; Get the pointer to the last character
        DCA     WORD            ; And save it in a non-autoindex location
        TAD     @WORD           ; Get the last character we saw
        TAD     [-073]          ; Was it a command seperator ??
```

```
        SNA CLA                 ; ????
         JMP     BOOTS1         ; Yes -- go execute another command

; See if this command needs to be repeated...
        STA                     ; Load the AC with -1
        TAD     REPCNT          ; And add to the repeat counter
        SPA                     ; Is the counter positive ??
         JMP     BOOTS          ; No -- go read anothter line
        DCA     REPCNT          ; Yes -- save the new count
        TAD     REPLOC          ; And get the location to start from
        DCA     L               ; Backup the command scanner
        JMP     BOOTS1          ; Then go execute this command again
        .TITLE  Command Error Processing


;   This routine is called when a syntax error is found in the command and it
; echo the part of the command which has already been scanned inside question
; marks (very much like TOPS-10 used to do!).  After that, the monitor is
; restarted (i.e. the stack is cleaned up and another prompt issued).
COMERR: CLA                     ; Ignore the contents of the AC
        DCA     @L              ; And mark the end of what was actually scanned
        .PUSHJ  @[TQUEST]       ; Type the first question mark
        TAD     [CMDBUF-1]      ; And point to the command line
        .PUSHJ  @[TASCIZ]       ; Echo that
        .PUSHJ  @[TQUEST]       ; Then type another question
        JMP     RESTA           ; Go restart the monitor


;   This routine prints an error message and then restarts the monitor.  Unlike
; nearly every other routine in the monitor this one is called via a JMS
; instruction rather than a .PUSHJ, and that so that the address of the error
; message can be passed in line, in the word after the JMS.
;
; CALL:
;       JMS     @ZERROR
;        <address of error message>
ERROR:  0                       ; enter here with a JMS
        CLA                     ; the AC is unknown here
        .PUSHJ  @[TQUEST]       ; always type a question mark
        TAD     @ERROR          ; pick up the address of the message
        .PUSHJ  @[OUTSTR]       ; and type that out too
        JMP     @ZRESTA         ; restart the monitor and read another command
        .TITLE  Get Next Command Character


;   This routine will get the next character from the command line.  If the
; character is lower case, it is folded to upper case.  If the character is a
; TAB, it is converted to a space.  And, if the character is  ";" or "!" (the
; start of a comment) it is converted to zero (end of line).  Finally, the
; character is returned in both the AC and location SAVCHR.
GET:    CLA                     ; Be sure the AC is safe to use
        TAD     @L              ; Get another character from the line
        TAD     [-"A"-40]       ; Compare this character to lower case A
        SMA                     ; ???
         JMP     GET1           ; It might be a lower case letter

; The character is not lower case -- check for a TAB, ; or !...
        TAD     ["A"+40-CHTAB]  ; Is this a tab character ??
        SNA                     ; ???
         TAD     [" "-CHTAB]    ; Yes -- convert it to a space
        TAD     [CHTAB-"!"]     ; No -- Is it a ! character ??
        SNA                     ; ???
         TAD     [-"!"]         ; Yes -- Convert it to a null
        TAD     ["!"-073]       ; Last chance -- is it a ; ??
        SNA                     ; ???
         TAD     [-073]         ; Yes -- convert that to zero too
        TAD     [073]           ; No -- restore the original character
        JMP     GET2            ; Then store the character and return

; Here if the character might be lower case...
GET1:   TAD     ["A"-"Z"]       ; Compare to the other end of the range
        SPA SNA                 ; ???
```

```
         TAD     [-40]          ; It's lower case -- convert to upper
        TAD     ["Z"+40]        ; Restore the correct character

; Store the character and return...
GET2:   DCA     SAVCHR          ; Remember this character
        TAD     SAVCHR          ; And also return it in the AC
        .POPJ                   ; And that's it
        .TITLE  Simple Lexical Functions


;   This routine will skip over any spaces in the command line and return the
; next non-space character in the AC and SAVCHR...

; Here to start skipping with the next character...
SPACMP: .PUSHJ  @ZGET           ; Get the next character

; Here to consider the current character and then skip...
SPACM0: CLA                     ; Be sure the AC is safe to use
        TAD     SAVCHR          ; And look at the current character
        TAD     ZMSPACE         ; Compare it to a space
        SNA CLA                 ; ???
         JMP     SPACMP         ; Keep going until we don't find one
        TAD     SAVCHR          ; Restore the character
        .POPJ                   ; And we're all done

;   This routine will examine the current character (in SAVCHR) or the next
; character (via GET) for end of line, which is stored as a null byte).  If
; it isn't the EOL, then COMERR is called and the current command is aborted,
; otherwise this routine just returns...

; Enter here to examine the next character...
EOLNXT: .PUSHJ  @ZGET           ; Load the next character
                                ; Then fall into the current character test

; Enter here to examine the current character...
EOLTST: .PUSHJ  @ZSPACM0        ; Allow blanks at the end of the line
        SZA CLA                 ; Is it the end of the line ??
         JMP     @ZCOMERR       ; No -- that's bad
        .POPJ                   ; Yes -- that's good

;   This routine will test either the current character (via SAVCHR) or the
; next character (via GET) to see if it's a space.  If it isn't, then it
; jumps to COMERR and aborts the current command...

; Enter here to examine the next character...
SPANXT: .PUSHJ  @ZGET           ; get the next character
                                ; and fall into SPATST...

; Enter here to examine the current character
SPATST: CLA                     ; don't require that the AC be cleared
        TAD     SAVCHR          ; get the current character
        TAD     ZMSPACE         ; and compare it to a space
        SZA CLA                 ; well??
         JMP     @ZCOMERR       ; not equal - this is a bad command line
        .POPJ                   ; it's a space

;   This routine will backup the command scanner so that the character
; just read will be read again with the next call to GET...
BACKUP: STA                     ; Load the AC with -1
        TAD     L               ; Then decrement the line pointer
        DCA     L               ; ...
        .POPJ                   ; That's all it takes

        .PAGE
        .TITLE  Call Routines in Field 1


;   This routine will allow a routine in field zero to simulate a .PUSHJ
; to a routine in field one.  Even better, when the routine in field one
; executes a .POPJ, the return will eventually be to field zero!  The
; contents of the AC are preserved both ways across the call.
;
```

```
;CALL:
;       JMS     @ZPUSHJ1        ; cross field call
;        <addr>                 ; address of a routine in field 1
;       <return here>           ; with the AC preserved across the call
;
PUSHJ1: 0                       ; call here with a JMS instruction
        DCA     PUSHAC          ; save the caller's AC for a minute
        TAD     @PUSHJ1         ; then get caller's argument
        DCA     F1ADDR          ; that's the address of the routine to call
        TAD     PUSHJ1          ; now get caller's return address
        IAC                     ; and skip over the argument
        .PUSH                   ; put that on the stack
        CLA                     ; (PUSH doesn't clear the AC!)
        TAD     [POPJ1]         ; the field one routine will return to
        .PUSH                   ;  ... POPJ1: in field one
        CLA                     ; ...
        TAD     PUSHAC          ; restore the original AC contents
        CXF     1               ; call with IF = DF = 1
        JMP     @.+1            ; and go to the code in field 1
F1ADDR: .BLOCK  1               ; gets the address of the field 1 routine
PUSHAC: .BLOCK  1               ; a temporary place to save the AC

;   When the routine in field one executes a .POPJ, it will actually return to
; the code at POPJ1: _in field one_ !!  Since we've also stacked our original
; caller's return address, the code at POPJ1 really only needs to do two
; things, a "CXF 0" to return to field zero, and then another .POPJ.
; Unfortunately, this code has to live in field one, so you won't find it
; here!
        .TITLE  RP Command -- Repeat


;   This command allows the rest of the command line to be repeated, and it
; accepts an optional argument which specifes the number of times to repeat,
; in decimal.   The range for this argument is 1 to 2047 and if it is omitted,
; it defaults to 2047.  Note that repeat commands may not be nested; a repeat
; command will cancel any previous repeat on the same command line.  Any error
; or a control-C will terminate the repetition prematurely.
REPEAT: .PUSHJ  @ZSPACMP        ; Get the next character
        SNA CLA                 ; Is it the end of the command ??
         JMP     REPEA1         ; Yes -- use the default count
        .PUSHJ  @ZBACKUP        ; No -- backup the scanner
        .PUSHJ  @[DECNW]        ; Then read a decimal number
        .PUSHJ  @ZEOLTST        ; Then test for the end of the line

; Set up the repeat counter...
        STA                     ; Subtract one from the user's
        TAD     WORD            ;  argument...
        SKP                     ; ...
REPEA1: NL3777                  ; If there's no argument, use 2047
        DCA     REPCNT          ; Set the repeat counter

; Set up the repeat pointer...
        TAD     L               ; Get the current line pointer
        DCA     REPLOC          ; Remember where to start from
        TAD     @REPLOC         ; Then examine the current character
        SNA CLA                 ; Is the repeat the last command on the line ??
         DCA     REPCNT         ; Yes -- this is all pointless after all
        .POPJ                   ; Then proceed with the next command
        .TITLE  TW and TP Commands - Set Terminal Width and Page


;   The TW command sets the terminal screen width to the value of its argument,
; which is a _decimal_ number.  Screen widths are limited to the range 32..255.
TWCOM:  .PUSHJ  @[DECNW]        ; read a decimal operand again
        .PUSHJ  @ZEOLTST        ; then check for the end of the line
        TAD     WORD            ; get the desired width
        SNA                     ; is it zero ??
         JMP     TWCOM1         ; yes -- that disables automatic returns
        TAD     [-32.]          ; compare it to 32
        SPA                     ; is it at least 32 ??
         JMP     TFILV          ; no -- ?ILLEGAL VALUE
        TAD     [32.-255.]      ; now compare it to 255
```

```
        SMA SZA                 ; it can't be bigger than that
         JMP     TFILV          ; but it is...
        TAD     [255.]          ; restore the original number
TWCOM1: DCA     WIDTH           ; and set the terminal width
        .POPJ                   ; then that's all

; Here if the parameter value is illegal...
TFILV:  JMS     @ZERROR         ; yes -- it isn't legal
         ERRILV                 ; ?ILLEGAL VALUE

;   The TP command sets the terminal page size to the value of its argument,
; in _decimal_.  Page sizes may range from 12 to 48, or zero.  A value of zero
; disables the automatic XOFF function completely.
TPCOM:  .PUSHJ  @[DECNW]        ; Read a decimal operand
        .PUSHJ  @ZEOLTST        ; And check for the end of the line
        TAD     WORD            ; Get the value he gave
        SNA                     ; Is it zero ??
         JMP     TPCOM1         ; Yes -- that is legal (to disable)
        TAD     [-12.]          ; Compare it to 12 lines
        SPA                     ; We have to have at least that many
         JMP     TFILV          ; No -- ?ILLEGAL VALUE
        TAD     [16.-48.]       ; Then compare it to 48
        SMA SZA                 ; Is it more than that ??
         JMP     TFILV          ; Yes -- that won't work, either
        TAD     [48.]           ; Restore the original number
TPCOM1: DCA     LENGTH          ; And set the new terminal length
        .POPJ                   ; ...
        .TITLE  VE Command - Show System Name and Version


;   This routine will type the name and version of the monitor.  It is called
; at startup, and by the VE command.
VECOM:  .PUSHJ  @ZEOLNXT        ; enter here for the VE command
HELLO:  JMS     @ZINLMES        ; type out the name of the system
         SYSNM1                 ; ...
        TAD     [VERSION]       ; get the present edit level
        .PUSHJ  @[TOCT3]        ; type that in octal
        JMS     @ZINLMES        ; say
         SYSNM2                 ;  "Checksum "
        TAD     @[ROMCK0]       ; get the checksum of ROM field 0
        .PUSHJ  @ZTOCT4S        ; type that and a space
        CDF     1               ; then do the same for ROM field 1
        TAD     @[ROMCK1]       ; ...
        CDF     0               ; ...
        .PUSHJ  @ZTOCT4S        ; this time end with a CRLF
        JMS     @ZINLMES        ; finally, type the system date
         SYSNM3                 ; ...
        .PUSHJ  @ZCRLF          ; finish that line
        JMS     @ZINLMES        ; then type the copyright notice
         SYSCRN                 ; ...
        JMP     @ZCRLF          ; finish that line and we're done
        .TITLE  H Command - Show Monitor Help

;   The H command generates a simple list of all monitor commands and a
; brief, one line, help message for each.  The whole function is driven
; by a table of help messages stored in field one - this table contains
; a list of pointers and each pointer points to the packed ASCII text of
; a single line of help.  We simply print each line and add a CRLF at the
; end.  It's done this way (as a table of lines) rather than as a single,
; huge, string with embedded CRLFs because the automatic XOFF (i.e. terminal
; page) processing is done via the CRLF routine.  Embedded CRLFs wouldn't
; automatically XOFF, and so most of the text would scroll right off the
; top of the CRT.
HELP:   .PUSHJ  @ZEOLTST        ; no arguments are allowed
        TAD     [HLPLST-1]      ; point to the list of help messages
        DCA     X3              ; in an auto index register
HELP1:  CDF     1               ; the help text and table lives in field 1
        TAD     @X3             ; get the next help message
        CDF     0               ; back to our field
        SNA                     ; end of list?
         .POPJ                  ; yes - we can quit now
        .PUSHJ  @[OUTSTR]       ; nope - type this string
```

```
        .PUSHJ  @ZCRLF          ; and finish the line
        JMP     HELP1           ; keep typing until we run out of strings

        .PAGE
        .TITLE  E and EP Commands -- Examine Main Memory or Panel Memory


;   The E command allows the user to examine the contents of memory in octal,
; either one word at a time or an entire range of addresses.  In the latter
; case a memory dump is printed with eight PDP-8 words per line.  It accepts
; several forms of operands, for example:
;
;       >E 1234         -> Examine location 1234 in the data field
;       >E 01234        -> Examime location 1234 of field zero
;       >E 41234        -> Examime location 1234, field 4
;       >E 71000-2000   -> Examine locations 1000 through 2000, field 7
;       >E 50000-77777  -> Examine location 0, field 5 thru 7777, field 7
;
;   The EP command is identical to E, except that panel memory is examined
; rather than main memory.

; Enter here for the EP command...
EPMEM:  STA                     ; set the PNLMEM flag
        SKP                     ; fall into the regular code
; Enter here for the E command...
EMEM:   CLA                     ; clear the PNLMEM flag
        DCA     PNLMEM          ; to reference main memory

; Both forms join up here...
EMEM0:  .PUSHJ  @ZRANGE         ; go read an address range
        SNL                     ; was there just one address ???
         JMP     EONE           ; yes -- just examine one location

; Fix up the address range for multiple word examines...
        TAD     LOW             ; get the low boundry
        AND     [7770]          ; round it down to a multiple of 8
        DCA     ADDR            ; then it becomes the starting address
        TAD     HIGH            ; get the ending address
        AND     [7770]          ; round it up to a multiple of 8
        TAD     ZK7             ; ...
        DCA     HIGH            ; and remember the last address to process

; Type out lines of 8 memory locations...
EMEM1:  .PUSHJ  @[TADDR]        ; type out the address of the next word
EMEM2:  .PUSHJ  @ZRDMEM         ; go read a word from main memory
        .PUSHJ  @ZTOCT4S        ; type the word in octal
        .PUSHJ  @ZTSTADR        ; have we done all the locations ??
        SZL                     ; are we there yet ???
         JMP    EMEM3           ; yes -- finish the line and return
        .PUSHJ  @ZNXTADR        ; no -- increment to the next address
        TAD     ADDR            ; get the current address
        AND     ZK7             ; is it a multiple of 8 ??
        SZA CLA                 ; ???
         JMP     EMEM2          ; no -- keep typing
        .PUSHJ  @ZCRLF          ; yes -- start on a new line
        JMP     EMEM1           ; and type the next address

; Here to examine a single memory location...
EONE:   .PUSHJ  @[TMEM]         ; type out the contents of memory
                                ; and fall into the next range

; Here when we've finished examining one range of addresses...
EMEM3:  .PUSHJ  @ZCRLF          ; finish the current line
        .PUSHJ  @ZBACKUP        ; backup the command line pointer
        .PUSHJ  @ZSPACMP        ;  ... and get the next character
        SNA                     ; is it the end of the line ??
         .POPJ                  ; yes -- just stop now
        TAD     [-","]          ; no -- check for a comma
        SZA CLA                 ; ????
         JMP     @ZCOMERR       ; this isn't legal
        JMP     EMEM0           ; yes -- do another operand
```

```
;   This routine will type the address and contents of the memory location
; indicated by registers ADDR and ADRFLD.
TMEM:   .PUSHJ  @[TADDR]        ; first type the address
        .PUSHJ  @ZRDMEM         ; then load the indicated word
        JMP     @ZTOCT4S        ; type it out and return
        .TITLE  D and DP Commands -- Deposit in Main Memory or Panel Memory


;   The D command allows the user to deposit one or more words in memory.  The
; general format is:
;
;       >D 60123 4567     -> Deposit 4567 into location 0123, field 6
;       >D 40000 1,2,3,4  -> Deposit 0001 into location 0, field 4, and 0002
;                            into location 1, field 4, and 0003 into location
;                            2, etc...
;
;   The DP command is identical to D, except that panel memory is chaged rather
; than main memory.  WARNING - there is no protection against changing the
; monitor when using this command, so it's up to the user to make sure the
; changes don't corrupt something important!

; Enter here for the DP command...
DPMEM:  STA                     ; set the PNLMEM flag
        SKP                     ; fall into the regular code
; Enter here for the D command...
DMEM:   CLA                     ; clear the PNLMEM flag
        DCA     PNLMEM          ; to reference main memory

; Both forms join up here...
        .PUSHJ  @[RDADDR]       ; Then read an address
        .PUSHJ  @[SPATST]       ; the next character has to be a space

; Read words of data and deposit them...
DMEM1:  .PUSHJ  @ZOCTNW         ; read an octal operand
        TAD     WORD            ; get the data we found
        .PUSHJ  @ZDANDV         ; write and verify it
        .PUSHJ  @ZNXTADR        ; advance to the next address
        .PUSHJ  @ZSPACM0        ; get the break character from OCTNW
        SNA                     ; was it the end of the line ??
         .POPJ                  ; yes, we're done...
        TAD     [-","]          ; no - it has to be a comma otherwise
        SZA CLA                 ; ????
         JMP     @ZCOMERR       ; bad command
        JMP     DMEM1           ; go read and deposit another word
        .TITLE  ER and DR Commands - Examine and Deposit in Registers


;   The ER command examines either a single register, when the register name
; is given as an argument, or all registers when no argument is given.  For
; example:
;
;       >ER AC  - examine the AC
;       >ER PC  - examine the PC
;       >ER     - print all registers
;
EREG:   .PUSHJ  @ZSPACMP        ; get the next non-space character
        SNA CLA                 ; is it the end of line?
         JMP     @[REGLSC]      ; yes - type all registers and return
        .PUSHJ  @ZBACKUP        ; nope - backup the command scanner
        .PUSHJ  @[NAMENW]       ; and go read the register name
        .PUSHJ  @ZEOLNXT        ; now we have to be at the end of line
        TAD     [ENAMES-1]      ; point to the name table
        .PUSHJ  @[MATCH]        ; find it and call a routine to print
        JMP     @ZCRLF          ; finish the line and we're done

;   The DR command deposits a value in a register, and both a register name and
; an octal value are required arguments.  For example:
;
;       >DR AC 7777     - set the AC to 7777
;       >DR SR 3345     - set the switch register to 3345
;
DREG:   .PUSHJ  @[NAMENW]       ; get the register name
```

```
        .PUSHJ  @[SPANXT]       ; the terminator has to be a space
        .PUSHJ  @ZOCTNW         ; read an octal number to deposit
        .PUSHJ  @ZEOLTST        ; followed by the end of the line
        TAD     [DNAMES-1]      ; point to the list of deposit names
        JMP     @[MATCH]        ; call the right routine and we're done

        .PAGE
        .TITLE  Deposit in Registers


; Here to deposit in the AC...
DAC:    TAD     WORD            ; Get his value
        DCA     UAC             ; And change the AC
        .POPJ                   ; Then that's all

; Here to deposit in the PC...
DPC:    TAD     WORD            ; The same old routine...
        DCA     UPC             ; ...
        .POPJ                   ; ...

; Here to deposit in the MQ...
DMQ:    TAD     WORD            ; ...
        DCA     UMQ             ; ...
        .POPJ                   ; ...

; Here to deposit in the PS...
DPS:    TAD     WORD            ; ...
        AND     [6277]          ; only these bits can actually change
        MQL                     ; save the new value for a minute
        TAD     UFLAGS          ; get the current status
        AND     [1500]          ; clear the complementary bits
        MQA                     ; or everything together
        DCA     UFLAGS          ; and update the PS
        .POPJ                   ; ...

; Here to deposit in the switch register...
DSR:    TAD     WORD            ; ...
        WSR                     ; Load the switch register
        .POPJ                   ; Then that's all

        .TITLE  Examine Registers


;   This routine is called to type out all the important internal registers.
; It is used by the ER, and SI commands, and after breakpoints, traps and
; halts.
REGLST: CLA                     ; be sure the AC is cleared
        .PUSHJ  TYPEPC          ; type the PC first
        .PUSHJ  TYPEPS          ; then the LINK
        .PUSHJ  TYPEAC          ; then the AC
        .PUSHJ  TYPEMQ          ; then the MQ
        .PUSHJ  TYPSP1          ; user stack pointer 1
        JMP     TYPSP2          ; and finally stack pointer 2

; The same as REGLST, but with a carriage return added...
REGLSC: .PUSHJ  REGLST          ; first type the registers
        JMP     @ZCRLF          ; type the carriage return and we're done

;   This routine types a register name followed by an octal register value.
; The latter is passed in the AC, and the register name is passed inline.
TYPRG4: 0                       ; enter here with a JMS instruction
        DCA     VALUE           ; save the register contents for a moment
        TAD     @TYPRG4         ; and get the address of the register name
        .PUSHJ  @[OUTSTR]       ; type that
        TAD     VALUE           ; get the contents of the register
        JMP     @ZTOCT4S        ; type that in octal and leave a blank

; This routine will type the last user AC contents...
TYPEAC: TAD     UAC             ; get the contents of the register
        JMS     TYPRG4          ; type it and return
         ACNAME                 ; "AC>"
```

```
; This routine will type the last user PC...
TYPEPC: TAD     UPC                 ; the same old routine...
        JMS     TYPRG4              ; ...
         PCNAME                     ; "PC>"

; This routine will type the last user MQ contents...
TYPEMQ: TAD     UMQ                 ; ...
        JMS     TYPRG4              ; ...
         MQNAME                     ; "MQ>"

; This routine will type the last instruction executed...
TYPEIR: TAD     UIR                 ; ...
        JMS     TYPRG4              ; ...
         IRNAME                     ; "IR>"

; This routine will type the current interrupt flags...
TYPEPS: TAD     UFLAGS              ; get the flags
        JMS     TYPRG4              ; ...
         PSNAME                     ; "PS>"

; This routine will type the 6120 stack pointer #1...
TYPSP1: TAD     USP1                ; ...
        JMS     TYPRG4              ; ...
         SP1NAM                     ; "SP1>"

; This routine will type the 6120 stack pointer #2...
TYPSP2: TAD     USP2                ; ...
        JMS     TYPRG4              ; ...
         SP2NAM                     ; "SP2>"

; This routine will type the current switch register contents...
TYPESR: LAS                         ; actually read the switch register
        JMS     TYPRG4              ; ...
         SRNAME                     ; "SR>"
        .TITLE  Read and Write Memory


;   This routine will change the current data field to the field indicated in
; location ADRFLD.  It's normally used by commands that read or write memory,
; such as Examine, Deposit, etc.  Remember that on the 6120 the EMA works in
; panel memory as well, so don't forget to change back to field zero after
; you're done!
CFIELD: CLA                         ; ...
        TAD     ADRFLD              ; get the desired field number
        AND     ZK70                ; just in case!
        TAD     [CDF 0]             ; and make a CDF instruction
        DCA     .+1                 ; store that in memory
        0                           ; isn't self manipulation wonderful?
        .POPJ                       ; that's all

;   This routine will set or clear the panel data flag according to the state
; of the PNLMEM flag.  If PNLMEM is non-zero, the panel data flag is set and
; commands that access memory (e.g. Examine, Deposit, etc) access panel memory
; instead.  If PNLMEM is zero, then the panel data flag is cleared and these
; commands access main memory.
CPANEL: CLA                         ; don't expect anything from the caller
        SPD                         ; assume we're referencing panel memory
        TAD     PNLMEM              ; but get the flag to be sure
        SNA CLA                     ; non-zero means access panel memory
         CPD                        ; we were wrong - use main memory instead
        .POPJ                       ; and we're done

;   This short routine returns, in the AC and memory location VALUE, the
; contents of the memory location addressed by ADDR and ADRFLD.  If PNLMEM is
; non-zero it reads panel memory to get the data; otherwise it reads main
; memory...
RDMEM:  .PUSHJ  CFIELD              ; first select the proper field
        .PUSHJ  CPANEL              ; then select main memory or panel memory
        TAD     @ADDR               ; load the data
        DCA     VALUE               ; save the contents in VALUE
        TAD     VALUE               ; and also return it in the AC
        SPD                         ; back to panel memory
```

```
        CDF     0               ; and back to the monitor's field
        .POPJ                   ; that's all there is

;   This routine will deposit the contents of the AC into the memory location
; specified by ADRFLD, ADDR and PNLMEM.  It's the complement of RDMEM...
WRMEM:  DCA     VALUE           ; save the number to deposit
        .PUSHJ  CFIELD          ; be sure we're in the right field
        .PUSHJ  CPANEL          ; and the right memory space (panel vs main)
        TAD     VALUE           ; get the value back again
        DCA     @ADDR           ; store the data
        SPD                     ; back to panel memory
        CDF     0               ; and the monitor's data field
        .POPJ                   ; and return

;   This routine is just like WRMEM, except that it will read back the value
; deposited and verify that it is, in fact, correct!  If it isn't (i.e. there's
; no memory at that address or the memory there isn't working) a ?MEM ERR
; message is generated and this command is aborted.
DANDV:  DCA     GOOD            ; save the original, good, value
        TAD     GOOD            ; ...
        .PUSHJ  WRMEM           ; store it
        .PUSHJ  RDMEM           ; read it back
        CIA                     ; make what we read negative
        TAD     GOOD            ; and compare it to the desired value
        SNA CLA                 ; did they match ??
         .POPJ                  ; yes -- this memory is ok

; Type a "?MEM ERR AT faaaa ..." message
        JMS     @ZINLMES        ; type out the first part of the message
         MEMMSG                 ; ...
        .PUSHJ  @[TADDR]        ; then type the address of the error
        TAD     GOOD            ; get the expected value
        .PUSHJ  @ZTOCT4S        ; type that out first
        .PUSHJ  @ZRDMEM         ; read what we actually get from memory
        .PUSHJ  @ZTOCT4C        ; then type that with a CRLF
        JMP     @ZRESTA         ; and bomb this command

; Temporary storage for DANDV...
GOOD:   .BLOCK  1               ; the "good" value we wrote to memory
        .PAGE
        .TITLE  BM Command -- Memory Block Move


;   The BM command is used to move blocks of memory words from one location to
; another.  It has three parameters - the source address range (two 15 bit
; numbers), and the destination address (a single 15 bit number).  All words
; from the starting source address to the ending source address are transferred
; to the corresponding words starting at the destination address.  More than
; one field may be transferred, and transfers may cross a field boundry.
;
;       >BM 200-377 400         -> move all of page 1 in the current data field
;                                  to page 2 of the same field.
;       >BM 0-7777 10000        -> move all of field 0 to field 1
;       >BM 00000-37777 40000   -> move fields 0 thru 3 to fields 4 thru 7
;
;   Note that this command operates only on main memory - there is no
; corresponding block move command for panel memory!
;
BMOVE:  .PUSHJ  @ZRANGE         ; read the source address range
        SNL                     ; did he give 2 numbers ???
         JMP     @ZCOMERR       ; no -- don't allow a one address range
        .PUSHJ  @[RDADDR]       ; now read the destination
        .PUSHJ  @ZEOLTST        ; this should be the end of the command
        DCA     PNLMEM          ; this command ALWAYS operates on main memory

; Now copy words from the source to the destination...
MOVE1:  .PUSHJ  @[SWPADR]       ; swap the LOW/LOWFLD (the source address)
        .PUSHJ  @ZRDMEM         ; read a word from the source
        .PUSHJ  @ZTSTADR        ; go see if this is the last address
        SZL CLA                 ; is it the end ???
         STA                    ; yes -- load -1 into the AC
        DCA     COUNT           ; and remember that fact for later
```

```
        .PUSHJ  @ZNXTADR        ; now increment the source address
        .PUSHJ  @[SWPADR]       ; swap the source back into LOW/LOWFLD
        TAD     VALUE           ; get the data we read from the source
        .PUSHJ  @ZDANDV         ; and deposit it in the destination
        .PUSHJ  @ZNXTADR        ; increment the destination address too
        SZL                     ; did we wrap out of field 7 ???
         JMP     MOVE2          ; yes -- stop here
        ISZ     COUNT           ; have we copied all the words ??
         JMP     MOVE1          ; no -- keep looping
        .POPJ                   ; yes -- that's all

; Here if the destination address runs out of field 7...
MOVE2:  JMS     @ZERROR         ; don't allow that to continue
         ERRWRP                 ; ?WRAP AROUND
        .TITLE  CK Command -- Checksum Memory


;   This command will compute the checksum of all memory locations between the
; two addresses specified and the resulting 12 bit value is then printed
; on the terminal.  This is useful for testing memory, comparing blocks of
; memory, and so on.  Note that the checksum algorithm used rotates the
; accumulator one bit between each words, so blocks of memory with identical
; contents in different orders will give different results.
;
;       >CK 10000-10177         -> checksum all of page 0, field 1
;
;   Note that this command operates only on main memory - there is no
; corresponding command for panel memory!
CKMEM:  .PUSHJ  @ZRANGE         ; read a two address range
        SNL                     ; two addresses are required
         JMP     @ZCOMERR       ; ...
        .PUSHJ  @ZEOLTST        ; be sure this is the end of the command
        DCA     PNLMEM          ; this command ALWAYS operates on main memory
        DCA     CHKSUM          ; and clear the checksum accumulator

; Read words and checksum them...
CKMEM1: TAD     CHKSUM          ; get the previous total
        CLL RAL                 ; and shift it left one bit
        SZL                     ; did we shift out a one ??
         IAC                    ; yes -- shift it in the right end
        DCA     CHKSUM          ; save that for a while
        .PUSHJ  @ZRDMEM         ; and go read a word from real memory
        TAD     CHKSUM          ; add it to the checksum
        DCA     CHKSUM          ; save the new checksum
        .PUSHJ  @ZTSTADR        ; compare the addresses next
        SZL                     ; are we all done ??
         JMP     TCKSUN         ; yes -- type the checksum and return
        .PUSHJ  @ZNXTADR        ; no -- increment the address
        JMP     CKMEM1          ; and proceed

;   This routine will type out the checksum currently contained in location
; CHKSUM.  If the checksum isn't zero, it will type a question mark (making
; a pseudo error message) first...
TCKSUM: TAD     CHKSUM          ; see if the checksum is zero
        SNA CLA                 ; ???
         JMP     TCKSUN         ; yes -- type it normally
        .PUSHJ  @[TQUEST]       ; no -- type a question mark first

; Now type the checksum...
TCKSUN: JMS     @ZINLMES        ; type out the checksum message
         CKSMSG                 ; ...
        TAD     CHKSUM          ; get the actual checksum
        JMP     @ZTOCT4C        ; type it with a CRLF and return
        .TITLE  CM and FM Commands -- Clear Memory and Fill Memory


;   The FM command fills a range of memory locations with a constant.  For
; example:
;
;       >FM 7402 0-7777         -> fill all of field zero with HLT instructions
;       >FM 7777 0-77777        -> fill all of memory with -1
;
```

```
;   The second and third arguments (the address range) may be omitted, in
; which case all of memory is filled.
;
;   Note that this command operates only on main memory - there is no
; corresponding command for panel memory!
FLMEM:  .PUSHJ  @ZOCTNW         ; read the constant to fill with
        TAD     WORD            ; get the desired value
        JMP     CMEM0           ; then join the CM command

;   The CM command is identical to FM, except that the fill value is always
; zero (hence the name - "Clear" memory).   For example:
;
;       >CM 50000 57777         -> clear all of field 5
;       >CM                     -> (with no arguments) clear all of memory!
;
;   Like FM, this command operates only on main memory.   There is no
; equivalent for panel memory.
CMEM:   .PUSHJ  @ZGET           ; advance the scanner to the break character
        CLA                     ; and throw it away for now
CMEM0:  DCA     VALUE           ; and fill with zeros
        DCA     ADDR            ; initialize the address range to start
        DCA     ADRFLD          ;  at location 0000, field 0
        STA                     ; and to finish at location 7777,
        DCA     HIGH            ; ...
        TAD     ZK70            ;  field 7
        DCA     HGHFLD          ; ...
        DCA     PNLMEM          ; this command ALWAYS operates on main memory

; See if there is an address range...
        .PUSHJ  @ZSPACM0        ; get the break character
        SNA CLA                 ; is there any more out there ??
         JMP     CMEM1          ; no -- start filling
        .PUSHJ  @ZBACKUP        ; yes - backup to the first character
        .PUSHJ  @ZRANGE         ; and read the address range
        .PUSHJ  @ZEOLTST        ; then check for the end of the line

; Clear/set memory locations...
CMEM1:  TAD     VALUE           ; get the value to store
        .PUSHJ  @ZDANDV         ; store it and verify
        .PUSHJ  @ZTSTADR        ; see if we have done all the addresses
        SZL                     ; well ??
         .POPJ                  ; yes -- we can stop now
        .PUSHJ  @ZNXTADR        ; no -- increment the address field
        JMP     CMEM1           ; then go clear this word

        .PAGE
        .TITLE  WS Command -- Word Search Memory


;   The WS command searches memory for a specific bit pattern.  It accepts up
; to 4 operands: (1) the value to search for, (2) the starting search address,
; (3) the final search address, and (4) a search mask.  All values except the
; first are optional and have appropriate defaults.  Any location in the
; specified range which matches the given value after being masked is typed
; out along with its address.  For example:
;
;       >WS 6031                -> search all of memory for KSF instructions
;       >WS 6031 30000-33777    -> search words 0..3377 of field 3 for KSFs
;       >WS 6030 0-77777 7770   -> search memory for any keyboard IOTs
;
;   N.B. this command operates only on main memory and there is no equivalent
; for panel memory.

; Read the first (required) operand and set defaults for all the rest...
SEARCH: .PUSHJ  @ZOCTNW         ; read the value to search for
        TAD     WORD            ; then get that
        DCA     KEY             ; and save it
        DCA     ADDR            ; set the starting address to 0
        DCA     ADRFLD          ; in field 0
        STA                     ; then stop at location 7777
        DCA     HIGH            ; ...
        TAD     ZK70            ; field 7
```

```
        DCA     HGHFLD          ; ...
        STA                     ; and set the mask to 7777
        DCA     MASK            ; ...
        DCA     PNLMEM          ; this command _always_ searches main memory

; Try to read any optional operands...
        TAD     SAVCHR          ; is there any more there ??
        SNA CLA                 ; ???
         JMP     SEAR1          ; no -- start looking
        .PUSHJ  @ZRANGE         ; yes -- read the address range
        TAD     SAVCHR          ; is there a mask out there ??
        SNA CLA                 ; ???
         JMP     SEAR1          ; no -- start looking
        .PUSHJ  @ZOCTNW         ; yes -- read the mask too
        TAD     WORD            ; load the mask
        DCA     MASK            ; and save that for later
        .PUSHJ  @ZEOLTST        ; this has to be the end of the line

; Here to start the search...
SEAR1:  DCA     COUNT           ; count the number of matches we find
        TAD     KEY             ; get the search key
        AND     MASK            ; apply the mask to it too
        CIA                     ; make it negative
        DCA     KEY             ; and remember that instead

; Look through memory for matches...
SEAR2:  .PUSHJ  @[INCHRS]       ; poll the operator for control-C
        .PUSHJ  @ZRDMEM         ; read a word from real memory
        AND     MASK            ; apply the mask to it
        TAD     KEY             ; and compare to the value
        SZA CLA                 ; does it match ??
         JMP     SEAR3          ; no -- skip it
        .PUSHJ  @[TMEM]         ; yes -- type the address and contents
        .PUSHJ  @ZCRLF          ; then finish the line
        STA                     ; make the AC non-zero
        DCA     COUNT           ; and remember that we found a match
SEAR3:  .PUSHJ  @ZTSTADR        ; see if we have looked everywhere
        SZL                     ; well ??
         JMP     SEAR4          ; yes -- finish up now
        .PUSHJ  @ZNXTADR        ; no -- increment the address
        JMP     SEAR2           ; and keep looking

; Here at the end of the search...
SEAR4:  TAD     COUNT           ; see how many matches there were
        SZA CLA                 ; were there any at all ??
         .POPJ                  ; yes -- that's fine
        JMS     @ZERROR         ; no -- give an error message
         ERRSRF                 ; ? SEARCH FAILS

; Temporary storage for the SEARCH routine...
KEY:    .BLOCK  1               ; a search key
MASK:   .BLOCK  1               ; a search mask

        .TITLE  BL Command -- List Breakpoints


;   This command will list all the breakpoints which are currently set in
; the user's program.  It has no operands...
;
;       >BL
;
BLIST:  .PUSHJ  @ZEOLNXT        ; there should be no more
        .PUSHJ  BSETUP          ; set up X1, X2 and COUNT
        DCA     VALUE           ; count the number of breakpoints here

; Loop through the breakpoint table and list all valid breakpoints...
BLIST1: TAD     @X1             ; get the address of this breakpoint
        DCA     ADDR            ; and remember that
        TAD     @X2             ; then get the corresponding field
        DCA     ADRFLD          ; ...
        TAD     ADDR            ; let's see that address again
        SNA CLA                 ; is there really a breakpoint set here ??
```

```
         JMP     BLIST2         ; no -- on to the next one
        .PUSHJ  @[TMEM]         ; yes -- type out the address and memory
        .PUSHJ  @ZCRLF          ; finish this line
        ISZ     VALUE           ; and count the number we find
BLIST2: ISZ     COUNT           ; are there more breakpoints to do?
         JMP     BLIST1         ; yes - keep going

; Here after going through the table...
        TAD     VALUE           ; see how many we found
        SZA CLA                 ; any at all ??
         .POPJ                  ; yes -- that's great
        JMS     @ZERROR         ; no -- print an error message
         ERRNBP                 ; ?NONE SET

;   This routine will set up the pointers for traversing the break point
; table.  X1 always points to the break point address table, X2 points to
; the break point field table, and X3 points to the break point data table.
; COUNT is initialized to the negative of the table size (all three tables
; are the same size) so that it can be used as an ISZ counter.  This same
; arrangement of pointers is used by all the routines that operate on
; breakpoints.
BSETUP: CLA                     ; just in case...
        TAD     [BPTADR-1]      ; X1 points to the address table
        DCA     X1              ; ...
        TAD     [BPTFLD-1]      ; X2 points to the field table
        DCA     X2              ; ...
        TAD     [BPTDAT-1]      ; X3 points to the data table
        DCA     X3              ; ...
        TAD     [-MAXBPT]       ; and COUNT is the table size
        DCA     COUNT           ; ...
        DCA     PNLMEM          ; break points always refer to main memory!
        .POPJ                   ; ...
        .TITLE  Search for Breakpoints


;   This routine will search the breakpoint table to see if there is a one set
; at the location specified by ADDR and  ADRFLD.  If one is found, it will
; return with the LINK set and with X1/X2 pointing to the table entry.  If
; there is no breakpoint at the specified address, it returns with the LINK
; cleared.
BPTFND: .PUSHJ  BSETUP          ; set up X1, X2 and COUNT

; Look through the entire table...
BPTFN1: TAD     @X1             ; get this breakpoint address
        SNA                     ; is there breakpoint here at all ??
         JMP    BPTFN2          ; no -- just forget it
        CIA                     ; make this address negative
        TAD     ADDR            ; and compare to what we want
        SZA CLA                 ; does it match ??
         JMP     BPTFN2         ; no -- on to the next one
        TAD     @X2             ; yes -- get the field number
        CIA                     ; make that negative
        TAD     ADRFLD          ; and compare to the field we need
        SZA CLA                 ; do they match to ??
         JMP     BPTFN3         ; no -- keep looking
        STL                     ; yes -- set the LINK
        .POPJ                   ; and stop right now

; Here if the current address dosen't match...
BPTFN2: ISZ     X2              ; increment the field pointer too
BPTFN3: ISZ     COUNT           ; have we searched the entire table?
         JMP     BPTFN1         ; no -- keep looking
        CLL                     ; yes -- clear the LINK
        .POPJ                   ; and return

        .PAGE
        .TITLE  BR Command -- Remove Breakpoints


;   The BR command removes a breakpoint at a specific address or, if no
; operand is given, removes all breakpoints.  For example:
;
```

```
;       >BR 17605       -> remove the breakpoint at location 7605, field 1
;       >BR             -> remove all breakpoints regardless of address
;
BREMOV: .PUSHJ  @ZGET           ; get the next character
        SNA CLA                 ; is it the end of the line ??
         JMP    BPTCLR          ; yes -- clear all breakpoints
        .PUSHJ  @ZBACKUP        ; no -- backup the scanner
        .PUSHJ  @[OCTNI]        ; then read an address
        TAD     WORD            ; get the breakpoint address
        SNA                     ; be sure it isn't location zero
         JMP     @ZCOMERR       ; that isn't legal
        DCA     ADDR            ; save the address of the breakpoint

; Here to remove a single breakpoint...
        .PUSHJ  @[BPTFND]       ; look for this breakpoint it the table
        SNL                     ; did we find it ??
         JMP     BREMO1         ; no -- there's no breakpoint at that address
        TAD     X1              ; yes -- get the pointer to BPTADR
        DCA     ADDR            ; and save it in a non-autoindex location
        DCA     @ADDR           ; clear the BPTADR entry (to remove it)
        .POPJ                   ; and that's all

; Here if the breakpoint does not exist...
BREMO1: JMS     @ZERROR         ; give an appropriate error message
         ERRNST                 ; ?NOT SET

; Here to clear all breakpoints...
BPTCLR: .PUSHJ  @[BSETUP]       ; setup X1, X2, X3 and COUNT
BPTCL2: DCA     @X1             ; clear this breakpoint
        DCA     @X2             ; ...
        DCA     @X3             ; ...
        ISZ     COUNT           ; have we done them all?
         JMP     BPTCL2         ; no -- keep looping
        .POPJ                   ; yes -- that's it
        .TITLE  BP Command -- Set Breakpoints


;   The BP command sets a breakpoint in the main memory (aka user) program.
; It requires a single argument giving the 15 bit address where the breakpoint
; is to be set.  For example:
;
;       >BP 07605       -> set a breakpoint at location 7605, field 0
;       >BP 7605        -> same, but in the current instruction field
;
;   It's not possible to set a breakpoint at location zero in any field because
; the monitor uses zero as a marker for an unused breakpoint table entry.
;
;   Note that this routine only enters the breakpoint into the table - nothing
; actually happens to the main memory program until we start running it and
; the BPTINS routine is called.
BPTCOM: .PUSHJ  @[OCTNI]        ; go read the address
        TAD     WORD            ; get the address operand
        SNA                     ; be sure it isn't zero
         JMP     @ZCOMERR       ; no breakpoints at address zero
        DCA     ADDR            ; and put it in a safe place
        .PUSHJ  @ZEOLTST        ; then test for the end of the line

; See if this breakpoint is already in the table..
        .PUSHJ  @[BPTFND]       ; ...
        SNL CLA                 ; was it found ??
         JMP    BPTCO1          ; no -- go try to add it
        JMS     @ZERROR         ; yes -- say that it is already set
         ERRAST                 ; ?ALREADY SET

; Here to search for a free location in the table...
BPTCO1: .PUSHJ  @[BSETUP]       ; setup X1 and COUNT
BPTCO2: ISZ     X2              ; keep X1 and X2 in sync
        TAD     @X1             ; get this table entry
        SNA CLA                 ; have we found an empty one ??
         JMP     BPTCO3         ; yes -- great
        ISZ     COUNT           ; have we searched the entire table?
         JMP     BPTCO2         ; no -- keep trying
```

```
        JMS     @ZERROR         ; yes -- say that the table is full
         ERRBTF                 ; ?TABLE FULL

; Here to insert the breakpoint in the table...
BPTCO3: TAD     X1              ; get the pointer to the free location
        DCA     LOW             ; and put it in a non-autoindex location
        TAD     ADDR            ; get the desired address
        DCA     @LOW            ; and store that in the table
        TAD     X2              ; do the same with the field table
        DCA     LOW             ; ...
        TAD     ADRFLD          ; get the field we need
        DCA     @LOW            ; and put that in the table
        .POPJ                   ; then that's all
        .TITLE  Insert Breakpoints in Memory


;   This routine will insert breakpoints in main memory (aka the user program)
; at the locations specified in the breakpoint table.  The current contents of
; each breakpoint location are stored in CP memory in the BPTDAT table, and
; then are replaced by a BPT instruction.  This routine is normally called
; just before returning control to the user's program.
BPTINS: .PUSHJ  @[BSETUP]       ; set up X1, X2, X3 and COUNT

; Loop through the table and insert the breakpoints...
BPTIN1: TAD     @X1             ; get the next address
        SNA                     ; is there a breakpoint set there ??
         JMP     BPTIN2         ; no -- proceed to the next one
        DCA     ADDR            ; yes -- save the address
        TAD     @X2             ; and get the field
        DCA     ADRFLD          ; save that too
        .PUSHJ  @ZRDMEM         ; go read the contents of that location
        DCA     @X3             ; save the user's data in the table
        TAD     [BPT]           ; then get a breakpoint instruction
        .PUSHJ  @ZDANDV         ; deposit that in the breakpoint location
        JMP     BPTIN3          ; proceed to the next one

; See if we have finished the table...
BPTIN2: ISZ     X2              ; keep the pointers in sync
        ISZ     X3              ; ...
BPTIN3: ISZ     COUNT           ; have we been all the way through ??
         JMP     BPTIN1         ; no -- keep going
        .POPJ                   ; yes -- quit now
        .TITLE  Remove Breakpoints from Memory


;   This routine will restore the original contents of all breakpoint locations
; in the main memory program from the table at BPTDAT.  It is normally called
; after a trap to CP memory occurs.  Breakpoints must be restored so that the
; user may examine or change them.
BPTRMV: .PUSHJ  @[BSETUP]       ; set up X1, X2, X3 and COUNT

; Loop through the breakpoint table and restore all data...
BPTRM1: TAD     @X1             ; get the address of this breakpoint
        SNA                     ; is there one there at all ??
         JMP     BPTRM2         ; no -- on to the next one
        DCA     ADDR            ; yes -- remember the address
        TAD     @X2             ; then get the correct field too
        DCA     ADRFLD          ; ...
        TAD     @X3             ; finally get the original contents
        .PUSHJ  @ZDANDV         ; deposit and verify it back where it goes
        JMP     BPTRM3          ; on to the next one

; Here to advance to the next breakpoint...
BPTRM2: ISZ     X2              ; keep the pointers in sync
        ISZ     X3              ; ...
BPTRM3: ISZ     COUNT           ; have we done them all ??
         JMP     BPTRM1         ; no -- keep looping
        .POPJ                   ; yes -- that's it for this time
        .TITLE  TR Command -- Single Instruction with Trace


;   The TR command will execute one instruction of the user's program and then
```

```
; print the registers.  It always executes one instruction, but it may be
; combined with the repeat (RP) command to execute multiple instructions.
SICOM:  .PUSHJ  @ZEOLNXT        ; there are no operands

; Figure out what we are going to execute...
        TAD     UFLAGS          ; get the instruction field
        AND     ZK70            ; ...
        DCA     ADRFLD          ; so we can change to that field
        TAD     UPC             ; get the current main memory PC
        DCA     ADDR            ; and point to that
        DCA     PNLMEM          ; always access main memory
        .PUSHJ  @ZRDMEM         ; go read what we're about to execute
        DCA     UIR             ; remember that for later

; Execute 1 instruction...
        .PUSHJ  @[SINGLE]       ; just like it says

; Print all the registers...
        .PUSHJ  @[TYPEIR]       ; first type the UIR
        JMP     @[REGLSC]       ; and then print the rest and return

        .PAGE
        .TITLE  SI and P Commands - Single Instruction and Proceed


;   This routine will execute a single instruction of the user's program and
; then return to the caller.  It is used directly to execute the SI command,
; and indirectly by many other commands...

; Here for the SI command...
SNCOM:  .PUSHJ  @ZEOLNXT        ; make sure that there is no more

;   Setting the HALT flip flop will cause the HM6120 to immediately trap back
; to panel mode after it has executed exactly one instruction of the main
; memory program.  This makes it easy to implement a single step function.
;
;   Note that SINGLE is a subroutine which you can actually call, via a
; .PUSHJ, from anywhere in the monitor and it will return after the main
; memory instruction has been executed.  This little bit of magic happens
; because the code at CONT1 saves the monitor stack and then restores
; it after the single instruction trap.
SINGLE: PGO                     ; first make sure the HALT flip flop is cleared
        HLT                     ; then make sure it's set
        STA                     ; set the software single step flag
        DCA     SIMFLG          ;  ... so that CPSAVE will know what to do
        JMP     CONT1           ; then restore the registers and go

;   The P command is used to proceed after the main memory program has stopped
; at a breakpoint.  You can't simply continue at this point because the PC
; points to the location of the breakpoint, and Continue would simply break
; again, instantly.  The Proceed command gets around this problem by first
; executing a single instruction, and then contining normally.
PROCEE: .PUSHJ  @ZEOLNXT        ; this command has no operands
        .PUSHJ  SINGLE          ; first execute the location under the BPT
        JMP     CONT            ; then restore the breakpoints and continue
        .TITLE  C Command - Restore Main Memory Context and Continue


;   This routine will restore all the user's registers and return to his
; program.  It is called directly for the continue command, and is used
; indirectly (to change contexts) by several other commands.
;
;   When this routine finishes in the most literal sense, the user mode
; program is running and the monitor is no longer active.  However the
; CONT function can and will actually return, via a POPJ, if the user
; program causes a breakpoint or single instruction trap.  This property is
; critical to the operation of the Proceed, TRace and Single Instruction
; commands!

; Here for the continue command...
CONTCM: .PUSHJ  @ZEOLNXT        ; continue has no operands
```

```
; Select free running mode and insert all breakpoints...
CONT:   .PUSHJ  @[BPTINS]       ; insert all breakpoints
        DCA     SIMFLG          ; clear our software single step flag
        PGO                     ; make sure the HALT flip-flop is cleared

;   Restore all registers and context switch.  Naturally, part of this involves
; restoring the original user mode stack pointers, so before we lose our own
; stack forever, we save a copy of the last monitor stack pointer in RAM.  It
; gets restored by the code at CPSAVE after a breakpoint or single instruction
; trap.
;
;   Another gotcha - if a transition on CPREQ L occurs while we're in panel
; mode, the BTSTRP flag in the panel status will still set anyway.  If that
; happens and we try to continue, the 6120 will trap back to panel mode
; immediately.  The simplest fix for this is to do a dummy read of the panel
; status flags, which clears them.
CONT1:  PRS                     ; dummy read of panel status to clear BTSTRP
        RSP1                    ; get our monitor's stack pointer
        DCA     STKSAV          ; and save it for later
        POST+0                  ; show post code 0
        TAD     USP1            ; reload stack pointer #1
        LSP1                    ; ...
        TAD     USP2            ; and stack #2
        LSP2                    ; ...
        TAD     UMQ             ; restore the MQ register
        MQL                     ; ...
        TAD     UFLAGS          ; restore the flags, including IF, DF and LINK
        RTF                     ; ...
        TAD     UAC             ; restore the AC
        PEX                     ; exit panel mode
        JMP     @UPC            ; and, lastly, restore the PC

;   At this point we're running the main memory program.  If that program
; causes a breakpoint or single instruction trap, then the HM6120 will enter
; the CPSAVE routine thru the vector at 7777.  After it figures out the reason
; for the trap, CPSAVE will restore the original monitor's stack, from STKSAV,
; and execute a .POPJ.  Only then will this routine "return".
        .TITLE  ST Command -- Start a Main Memory Program


;   The start command initializes the CPU registers and all I/O devices and
; then transfers control to a main memory (user) program.  A single, optional,
; argument may be given to specify the start address of the the main memory
; program.  If the start address is omitted, then the default is location
; 7777 of field 0 (this is a little strange by PDP-8 standards, but it's the
; typical reset vector for 6100 and 6120 devices).  For example:
;
;       >ST 00200       - start at location 200 in field 0 (DF is also 0)
;       >ST 70200       - start at location 200 in field 7 (DF is also 7)
;       >ST             - start at location 7777 of field 0 (DF is also 0)
;
START:  .PUSHJ  @ZGET           ; get the next character
        SNA CLA                 ; is there anything out there ??
         JMP     START1         ; no -- use the defaults

; Start at a specific (non-default) address...
        .PUSHJ  @ZBACKUP        ; backup to the start of the address
        .PUSHJ  @[OCTNI]        ; then read it
        .PUSHJ  @ZEOLTST        ; now it has to be the end of the line
        .PUSHJ  CLRCPU          ; clear the saved main memory registers
        TAD     WORD            ; and overwrite the PC with the desired address
        DCA     UPC             ; ...
        TAD     ADRFLD          ; get the start field
        CLL RTR                 ; and make the DF be the same
        CLL RTL                 ; ...
        TAD     ADRFLD          ; ...
        DCA     UFLAGS          ; those are the default processor flags
        JMP     CONT            ; insert any breakpoints and then go

; Start at the default address.
START1: .PUSHJ  CLRCPU          ; set all main saved CPU registers to default
        JMP     CONT            ; and then start there
```

```
        .TITLE  MR Command - Master Reset


;   The MR command executes a CAF instruction to assert IOCLR L and initialize
; all external I/O devices, and then it resets the saved state of the main
; memory program to the "default" values.  From the point of view of an I/O
; device on the bus, this is equivalent to pressing the RESET button, but it
; doesn't actually reset the CPU itself (which would re-initialize this
; monitor!).  This command doesn't have any effect on the contents of main
; memory.
CLRCOM: .PUSHJ  @ZEOLNXT        ; There are no operands

; Initialize the saved user context...
        .PUSHJ  CLRCPU          ; clear UAC, UPC, UMQ, etc...

;   Execute a CAF instruction to clear all I/O devices.  On the IM6100 we
; couldn't do this, since CAF would also clear the CP flag (!), but the 6120
; designers allowed for this case.
;
;   Unfortunately IOCLR L also resets the console UART, which plays havoc
; with any character that we might be transmitting at the moment.  The only
; safe thing is to wait for the console to finish before executing the CAF.
; Note that this will _leave_ the console flag cleared, which is the way a
; real PDP-8 program should expect it (clearing a real PDP-8 clears the
; console flag too, after all).
        TSF                     ; has the console finished yet?
        JMP     .-1             ; no - wait for it
        CAF                     ; clear all I/O flags

; Reset the IDE disk too.  If none is attached, then this is harmless...
        JMS     @ZPUSHJ1        ; (cross field call)
         IDEINI                 ; reset the IDE disk and then return
        JMS     @ZPUSHJ1        ; (cross field call)
         INIPMP                 ; and initialize the partition map
        .POPJ                   ; all done ...


;   This routine is called by the START and RESET commands and at system
; initialization to clear the saved user context...
CLRCPU: CLA                     ; start with all zeros
        DCA     UAC             ; clear the AC
        DCA     UMQ             ; and the MQ
        DCA     UFLAGS          ; the DF, IF and LINK
        DCA     USP1            ; stack pointer #1
        DCA     USP2            ; and #2
        STA                     ; then finally set the PC to 7777
        DCA     UPC             ; ...
        .POPJ                   ; ...
        .TITLE  EX Command - Execute IOT Instructions


;   The EX command allows a user to type in and execute an IOT instruction
; directly from the terminal, which can be very useful for testing peripheral
; devices.  Either one or two operands are allowed - the first is the octal
; code of the IOT to be executed, and the second (which is optional) is a
; value to be placed in the AC before the IOT is executed.  If it is omitted,
; zero is placed in the AC.  After the instruction is executed, the word SKIP
; is typed if the instruction skipped, along with the new contents of the AC.
;
;       >EX 6471        -> execute IOT 6741  (the AC will be cleared)
;       >EX 6474 1176   -> put 1176 in the AC and execute IOT 6474
;
;   WARNING - some care must be exercised with this command, since executing
; the wrong IOT can crash the monitor!
XCTCOM: .PUSHJ  @ZOCTNW         ; go read the IOT instruction code
        TAD     WORD            ; then get the value
        DCA     XCTBLK          ; save that where we'll execute it
        DCA     WORD            ; assume to use zero in the AC
        TAD     SAVCHR          ; next get the break character
        SNA CLA                 ; was it the end of the line ??
         JMP     XCT1           ; yes -- default to zero
        .PUSHJ  @ZOCTNW         ; no -- read another number
```

```
        .PUSHJ  @ZEOLTST        ; then test for the end of the line

; Be sure the instruction is really an IOT...
XCT1:   TAD     XCTBLK          ; get the instruction
        TAD     [-6000]         ; compare it to 6000
        SMA CLA                 ; is it an IOT or OPR instruction ?
         JMP     XCT2           ; yes -- that's OK
        JMS     @ZERROR         ; no -- don't allow this
         ERRILV                 ; ?ILLEGAL VALUE

; Execute the instruction...
XCT2:   DCA     COUNT           ; will be non-zero if the IOT doesn't skip
        TAD     WORD            ; get the value we're supposed to put in the AC
XCTBLK: NOP                     ; gets overwritten with the IOT to execute
         ISZ     COUNT          ; set the flag if it doesn't skip
        DCA     VALUE           ; and remember what is left in the AC

; Print the results of the instruction..
        TAD     COUNT           ; see if it skipped
        SZA CLA                 ; well ??
         JMP     XCT3           ; no -- no message
        JMS     @ZINLMES        ; yes -- say that it did
         SKPMSG                 ; ...
XCT3:   JMS     @ZINLMES        ; then print the AC after the instruction
         ACNAME                 ; ...
        TAD     VALUE           ; ...
        JMP     @ZTOCT4C        ; in octal, with a CRLF, and return

        .PAGE
        .TITLE  OS/8 Bootstrap


;   How to boot OS/8 (there's lots of documentation on how to write a device
; handler, even a system handler, but I couldn't find a description of how
; to make a bootable device anywhere!):
;
;   The primary bootstrap for a device (the one which you have to toggle in
; with the switches!) normally loads cylinder 0, head 0, sector 0 (which is
; the equivalent to OS/8 logical block zero) into memory page zero field zero.
; The code loaded into page zero is then started in some device specific way,
; but usually the primary bootstrap is overwritten by this data and the CPU
; just "ends up" there.
;
;   The first few words of block zero are called the secondary bootstrap, and
; it's normally found in the header for the system device handler.  OS/8 BUILD
; copies this code from the handler to block zero when it builds the system
; device.  The second half of block zero contains the system handler, what's
; resident in page 7600 while OS/8 is running, plus some OS/8 resident code
; that BUILD wraps around it.  All of the second half of block zero must be
; loaded into page 7600, field 0 by the secondary bootstrap.
;
;  The remainder of the first half of block zero, the part after the secondary
; bootstrap, contains the OS/8 resident code for field 1.  This starts some
; where around offset 47 in the first half of block zero, and this code needs
; to be loaded into the corresponding locations of page 7600, field 1.  The
; remaining words in page 7600, field 1 (i.e. those that belong to the
; secondary bootstrap) OS/8 uses for tables and their initial values are
; unimportant.  It suffices to simply copy all of the first half of block zero
; to page 7600, field 1.
;
;   All this discussion presupposes a single page system handler, as we have
; here.  For a two page handler BUILD will put the second page in the first
; half of block 66 on the system device and I believe (although I can't
; guarantee it) that the second half of this block also contains an image
; of the OS/8 resident code at page 7600, field 1.  This would make it the
; same as, excepting the bootstrap part, the first half of block zero.  In
; the case of a two page handler, the secondary bootstrap is also responsible
; for loading the second handler page from block 66 into page 7600, field 2.
; OS/8 bootstrap code (secondary bootstrap).
;
;   Once everything has been loaded, the secondary bootstrap can simply do a
; "CDF CIF 0" and then jump to location 7600 to load the keyboard monitor.
```

```
;
;   The primary bootstrap for the SBC6120 RAM and IDE disks are six words
; loaded in locations 0 thru 5:
;
;       0000/ 6206      PR0             / execute a panel request
;       0001/ 0001       1              /  1 for RAM disk, 4 for IDE disk
;       0002/ 0100       0100           /  read one page into field zero
;       0003/ 0000       0000           /  location zero
;       0004/ 0000       0000           / from page/block zero of the device
;       0005/ 7402      HLT             / should never get here
;
;   If all goes well, the HLT in location 5 is never executed - it gets
; overwritten by the secondary bootstrap code before the ROM returns from
; the PR0 function.
;
;   The B (BOOT) command in BTS6120 actually bypasses the primary bootstrap
; step and simply reads block zero of the boot device into page zero, field
; zero directly.  The VM01 and ID01 secondary bootstraps all contain a special
; "key" in words 0 thru 4, the ones which would normally overwrite the primary
; boostrap, and BTS6120 looks for this key to identify a bootable volume.
; If the key is found, then BTS6120 simply jumps to main memory location 5
; to start the secondary bootstrap and finish loading OS/8.
;
;   This system should also work for any other, non OS/8, system provided that
; it uses the same primary bootstrap shown above and that its secondary boot
; contains the special key in the first five words.  As long as the secondary
; bootstrap starts at offset 5, the remainder of its code is unimportant to
; BTS6120 and it can do anything it likes.
        .TITLE  Boot Sniffer


;   The secondary bootstraps for both VM01 and ID01 devices contain a special
; key, the ASCIZ string "BOOT", in the first five words.  The caller is
; expected to read block zero of the boot device into memory, and then call
; this routine to examine page zero, field zero, of main memory to determine
; if a valid secondary bootstrap exists.  If the key is found, then the LINK
; will be cleared on return...
CKBOOT: TAD     [BOOKEY-1]      ; point X1 to the key string
        DCA     X1              ; ...
        NL7777                  ; and point X2 to page zero of main memory
        DCA     X2              ; ...
CKBOO1: CLL                     ; be sure the LINK is in a known state
        TAD     @X1             ; get the next word of the key
        SNA                     ; have we done them all ?
         .POPJ                  ; yes - return success
        CIA                     ; make it negative
        CPD                     ; address main memory now
        TAD     @X2             ; and compare our key to what's there
        AND     ZK177           ; (PAL8 likes to set the high bit for ASCII!)
        SPD                     ; (back to panel memory)
        STL                     ; assume that we failed
        SZA CLA                 ; did the key match ?
         .POPJ                  ; nope - return with the LINK set
        JMP     CKBOO1          ; yes - keep testing...

; Ok, here it is - the magic key that makes a volume bootable!
BOOKEY: .ASCIZ  /BOOT/
        .TITLE  B Command - Boot Disk


;   The B command boots, or at least it tries to, either RAM or IDE disk.
; It can be used with an argument to specify the device to be booted, or
; without to ask BTS6120 to search for a bootable volume.  For example:
;
;       >B VM   - boot device VMA0
;       >B ID   - boot device IDA0
;       >B      - search VMA0, then IDA0, for a bootstrap
;
; If no valid bootsrap can be found, then the message "?Not bootable" is
; printed.
;
;   NOTE: It is currently only possible to bootstrap from unit zero for RAM
```

```
; disk, or partition zero in the case of IDE disk.
BOOT:   .PUSHJ  @ZSPACMP        ; get the next non-space character
        SNA CLA                 ; is it the end of the line ?
         JMP     BOOT1          ; yes - go search for a bootstrap
        .PUSHJ  @ZBACKUP        ; nope - backup to the first letter
        .PUSHJ  @[NAMENW]       ; and read the boot device name
        .PUSHJ  @ZEOLNXT        ; now there has to be an EOL

; Here if a specific device name is given on the command line...
        TAD     [BNAMES-1]      ; point to the table of boot names
        .PUSHJ  @[MATCH]        ; go call the right boot routine
        SZL                     ; did we find a bootstrap ?
         JMP     NOBOOT         ; nope - print an error message

; Now do the equivalent of a "ST 5" to start the bootstrap running...
BOOTGO: .PUSHJ  @[CLRCPU]       ; clear all saved main memory state
        TAD     [5]             ; the secondary bootstrap starts at offset 5
        DCA     UPC             ; ...
        JMP     @[CONT]         ; cross your fingers!

; Here to search for a bootable device...
BOOT1:  .PUSHJ  BTVMA0          ; first try booting VMA0
        SNL                     ; did we succeed?
         JMP     BOOTGO         ; yes - go start the bootstrap
        .PUSHJ  BTIDA0          ; nope - try IDA0 next
        SNL                     ; how about this?
         JMP     BOOTGO         ; yes - use that on instead

; Here if no bootstrap can be found...
NOBOOT: JMS     @ZERROR         ; print an error and return to command level
         ERRNBT                 ; ?NO BOOTSTRAP


; Here to attempt booting VMA0...
BTVMA0: JMS     @ZPUSHJ1        ; (cross field call)
         RDBOOT                 ; RAM disk primary bootstrap
        .PUSHJ  CKBOOT          ; is this volume bootable?
        SZL                     ; skip if yes
         .POPJ                  ; not bootable - just give up
        JMS     @ZINLMES        ; say
         VMAMSG                 ;  "-VMA0"
        .PUSHJ  @ZCRLF          ; ...
        CLL                     ; be sure to return success
        .POPJ                   ; ...


; Here to attempt booting IDA0...
BTIDA0: JMS     @ZPUSHJ1        ; (cross field call)
         IDBOOT                 ; IDE disk primary bootstrap
        .PUSHJ  CKBOOT          ; is this volume bootable?
        SZL                     ; skip if yes
         .POPJ                  ; not bootable - just give up
        JMS     @ZINLMES        ; say
         IDAMSG                 ;  "-IDA0"
        .PUSHJ  @ZCRLF          ; ...
        CLL                     ; and be sure to return success
        .POPJ                   ; ...
        .TITLE  Parse FORMAT Unit/Partition Argument


;   This routine will parse the unit/partition number argument for FORMAT.
; Since this command is little on the dangerous side (it does erase all the
; data on the disk, after all!), we'll go to the extraordinary length of
; asking for confirmation before we do anything.  Confirmation is nothing
; more than a single character (we don't wait for a carriage return) - "Y"
; continues with the format and anything else,  including ^C, aborts...
;
;   Assuming the user confirms, then the unit/partition number will be
; returned in the AC.  If the user aborts, or if there are any syntax
; errors, then we restart the command scanner and never return.
FMTARG: .PUSHJ  @ZOCTNW         ; read the unit/partition number
        .PUSHJ  @ZEOLTST        ; that has to be followed by the end of line
```

```
        JMS     @ZINLMES        ; say
         FCFMSG                 ;  "Format partition/unit "
        TAD     WORD            ; get the partition number once again
        DCA     VALUE           ; TOCT corrupts WORD,
        TAD     VALUE           ;  .... so we have to stash it here
        .PUSHJ  @ZTOCT4S        ; and type it out
        .PUSHJ  @[CONFRM]       ; go wait for a "Y" or "y"
        SNL CLA                 ; did he confirm?
         JMP     @ZRESTA        ; no - just abort now
        TAD     VALUE           ; yes he did - return the unit in the AC
        .POPJ                   ; ...

; Here if the RAM disk unit number is illegal...
NOUNIT: JMS     @ZERROR         ; ...
         ERRILV                 ; "?Illegal value"

;   This little routine verifies that a hard disk is attached to the system.
; If there is none, then an error message is printed and the command aborted.
NODISK: CDF     1               ; there's a disk attached
        TAD     @[DKSIZE]       ;  ... only if DKSIZE != 0
        CDF     0               ; ...
        SZA CLA                 ; skip if there's no disk there
         .POPJ                  ; yes - return now
        JMS     @ZERROR         ; print a message and abort the command
         ERRNDK                 ; "?No disk"

        .PAGE
        .TITLE  PM Command - Show and Edit Disk Partition Map


;   The PM command allows the default mapping of OS/8 units to IDE disk
; partitions to be changed.  PM accepts two arguments, both of which are
; optional.  The first argument is the OS/8 logical unit number, and the
; second argument a partition number, in octal.  Used without any arguments,
; the PM command will display a list of all eight OS/8 units and their current
; mappings.  With one argument, PM will display only the mapping for that
; unit, and with two arguments PM will change the mapping of that unit.
;
;       >PM u pppp      - map OS/8 ID01 unit u to IDE partition pppp
;       >PM u           - display the mapping for unit u
;       >PM             - display the mapping for all units
;
PMEDIT: .PUSHJ  @ZSPACMP        ; get the next non-space character
        SNA CLA                 ; is it the end of the line ?
         JMP     PMALL          ; yes - show the entire map
        .PUSHJ  @ZBACKUP        ; nope - backup and read this character
        .PUSHJ  @ZOCTNW         ; it should be a unit number
        TAD     WORD            ; get the value we read
        AND     [7770]          ; and the unit number must be less than 7
        SZA CLA                 ; ??
         JMP     PMEDI1         ; nope - "Illegal value"
        TAD     WORD            ; transfer the unit number
        DCA     COUNT           ; to COUNT for later use

; See if there's also a partition number on the line...
        .PUSHJ  @ZSPACM0        ; get the next non-space character
        SNA CLA                 ; is it the end of the line?
         JMP     PMSHOW         ; yes - print the mapping for this unit only
        .PUSHJ  @ZBACKUP        ; nope - read what comes next
        .PUSHJ  @ZOCTNW         ; this should be the partition number
        .PUSHJ  @ZEOLTST        ; and then we have to be at the EOL

; Here to change the mapping for a specific unit...
        TAD     COUNT           ; get the unit number
        TAD     [PARMAP-1]      ; and make an index into the mapping table
        DCA     X1              ; ...
        CDF     1               ; PARMAP lives in field one
        TAD     WORD            ; get the desired mapping
        DCA     @X1             ; and update the partition table
        CDF     0               ; back to this field
        .POPJ                   ; and we're all done
```

```
; Here if the unit number is illegal...
PMEDI1: JMS     @ZERROR         ; say
         ERRILV                 ;  "?Illegal value" and abort


; Here to show all eight entries in the entire partition map...
PMALL:  DCA     COUNT           ; start with unit zero
        .PUSHJ  PMSHOW          ; and show the mapping for that unit
        ISZ     COUNT           ; now onto the next one
        TAD     COUNT           ; have we done eight ?
        TAD     [-10]           ; ???
        SZA CLA                 ; well ?
         JMP    PMALL+1         ; nope - keep going
        .POPJ                   ; yes, we can quit now

; Here to show the mapping for the unit in COUNT...
PMSHOW: JMS     @ZINLMES        ; say
         PM1MSG                 ;  "Unit "
        TAD     COUNT           ; get the selected unit
        .PUSHJ  @[TDIGIT]       ; and type it
        JMS     @ZINLMES        ; now say
         PM2MSG                 ;  " mapped to partition "
        TAD     COUNT           ; get the count again
        TAD     [PARMAP-1]      ; and index the partition table
        DCA     X1              ; ...
        CDF     1               ; the partition table lives in field 1
        TAD     @X1             ; get the partition mapped to this unit
        CDF     0               ; ...
        JMP     @ZTOCT4C        ; type it, in octal, and a CRLF
        .TITLE  Disk Formatter, Pass 1


;   Pass one of the RAM/IDE disk formatter writes every block with a simple
; test pattern consisting of alternating words filled with the block number
; and its complement.  Although it's not too creative, this pattern does do
; two things - it guarantees that each block is unique (so we can make sure
; the disk addresssing is working!) and it does ensure that every bit gets
; tested with a zero and a one (so we can make sure the data lines are
; working).
;
;   This routine expects that a number of memory locations will be initialized
; before it's called.  RECSIZ must contain the negative of the logical record
; size for the device (-256 for IDE disk or -128 for RAM disk).  FMTCNT should
; contain the negative of the device size, in blocks/pages, and FMTWRP (a
; location in this routine!) must be initialized with the address of the
; disk write routine...
FMTP1:  JMS     @ZINLMES        ; say
         FM1MSG                 ;  ... "Writing "
        CDF     1               ; ...
        DCA     @ZDKRBN         ; reset the current disk block
        DCA     @ZRDPAGE        ; and page numbers

; Fill the disk buffer with the test pattern...
FMTP11: TAD     RECSIZ          ; get the negative of the record size
        CLL CML RAR             ; and divide it by two
        DCA     COUNT           ; since we'll fill the buffer in word pairs
        TAD     [DSKBUF-1]      ; point X1 to the disk buffer
        DCA     X1              ; ...
        CDF     1               ; (the disk buffer is in field 1)
FMTP12: TAD     FMTCNT          ; get the current block/page number
        DCA     @X1             ; store that
        TAD     FMTCNT          ; then store its complement
        CMA                     ; ...
        DCA     @X1             ; in the next word
        ISZ     COUNT           ; have we done the whole buffer?
         JMP     FMTP12         ; nope - keep filling
        CDF     0               ; return to our default field

; Write the buffer to the disk...
        JMS     @ZPUSHJ1        ; (cross field call)
         PNLBUF                 ; setup our temporary buffer in field 1
        TAD     RECSIZ          ; pass the record size to the I/O routine
```

```
        JMS     @ZPUSHJ1        ; (cross field call)
FMTWRP:  .BLOCK  1              ; modified to either DISKWR or RAMDWR
        SZL                     ; were there any errors ?
         JMP     @[DIOERR]      ; yes - quit now

; See if we've done the whole disk...
        CLA                     ; ...
        ISZ     FMTCNT          ; increment the page/block counter
         SKP                    ; not done yet - keep going
          .POPJ                 ; all done!
        CDF     1               ; disk data lives in field 1
        ISZ     @ZDKRBN         ; increment the disk block number
        ISZ     @ZRDPAGE        ; and the RAM disk page number
        CDF     0               ; back to safe ground

; Print a dot every so often to make a simple "progress bar"...
        TAD     RECSIZ          ; get the current record size
        CMA                     ; and make it a mask for the lower bits
        AND     FMTCNT          ; apply it to the current block/page number
        SZA CLA                 ; ...
         JMP     FMTP11         ; not time for a dot yet
        .PUSHJ  @[TDOT]         ; print a dot to show our progress
        JMP     FMTP11          ; and another page or block

        .PAGE
        .TITLE  Disk Formatter, Pass 2


;   Pass two of the RAM/IDE disk formatter reads back every block and verifies
; that the test pattern written by pass 1 is there.  If any block doesn't
; contain the data we expect, then a "Verification error" message will be
; printed, but verification continues.  This routine expects all the same
; data to be set up in FMTCNT and RECSIZ as Pass 1, and in addition it
; expects FMTRDP to be initialized with the address of the disk read routine.
FMTP2:  JMS     @ZINLMES        ; say
         FM2MSG                 ;  "Verifying"
        CDF     1               ; ...
        DCA     @ZDKRBN         ; reset the current disk block
        DCA     @ZRDPAGE        ; and page numbers
        CDF     0               ; ...

; Read the next block/page from the disk...
FMTP21: JMS     @ZPUSHJ1        ; (cross field call)
         PNLBUF                 ; setup a temporary disk buffer in panel memory
        TAD     RECSIZ          ; pass the record size to DISKRD
        JMS     @ZPUSHJ1        ; (cross field call)
FMTRDP:  .BLOCK  1              ; gets modified to either DISKRD or RAMDRD!
        SZL                     ; any I/O errors ?
         JMP     @[DIOERR]      ; yes - just give up now

; Verify that the data in the buffer matches what we wrote...
        TAD     RECSIZ          ; and get the block/page size
        CLL CML RAR             ; divide it by two
        DCA     COUNT           ; because we'll test in double word pairs
        TAD     [DSKBUF-1]      ; point X1 to the disk buffer
        DCA     X1              ; ...
        CDF     1               ; (disk buffer lives in field 1)
FMTP22: TAD     FMTCNT          ; get the current block/page number
        CIA                     ; make it negative
        TAD     @X1             ; and compare to the first word in the buffer
        SZA CLA                 ; it matches, no?
         JMP     FMTP29         ; no - verify error!
        TAD     @X1             ; the second word is the complement of the page
        TAD     FMTCNT          ; so that plus this
        IAC                     ; plus 1 should be zero!
        SZA CLA                 ; are we right?
         JMP     FMTP29         ; no - verify error!
        ISZ     COUNT           ; have we done the whole buffer?
         JMP     FMTP22         ; nope - keep testing
        CDF     0               ; return to our regular field

; See if we've done the whole disk...
```

```
FMTP23: ISZ     FMTCNT          ; increment the page/block counter
         SKP                    ; not done yet - keep going
          .POPJ                 ; all done!
        CDF     1               ; disk data lives in field 1
        ISZ     @ZDKRBN         ; increment the disk block number
        ISZ     @ZRDPAGE        ; and the RAM disk page number
        CDF     0               ; back to safe ground

; Print a dot every so often to make a simple "progress bar"...
        TAD     RECSIZ          ; get the current record size
        CMA                     ; and make it a mask for the lower bits
        AND     FMTCNT          ; apply it to the current block/page number
        SZA CLA                 ; ...
         JMP     FMTP21         ; not time for a dot yet
        .PUSHJ  @[TDOT]         ; print a dot to show our progress
        JMP     FMTP21          ; and another page or block

; Here if one (or more) words don't match..
FMTP29: CDF     0               ; restore the usual field
        .PUSHJ  @ZCRLF          ; we're in the middle of a line now
        JMS     @ZINLMES        ; so start a new one and print
         ERRDSK                 ;  "?Verification error, block/page "
        CDF     1               ; (disk data is in field 1)
        TAD     @ZDKRBN         ; get the current block/page number
        CDF     0               ; ...
        .PUSHJ  @ZTOCT4C        ; and type it (in octal)
        JMP     FMTP23          ; better luck with the next block
        .TITLE  DF Command - Format IDE Disk Partition


;   The DF command will "format" an IDE disk partition. The name is a misnomer
; because there's nothing about an IDE disk that needs formatting in the way
; a floppy does, but this command does write and then read back every single
; block of the partition which serves the useful function of testing the disk.
; It works in two passes - the first pass writes every block with a test
; pattern, and the second pass reads and verifies every block for the correct
; data.
;
;       >DF pppp        - format disk partition pppp
;
DFRMAT: .PUSHJ  @[NODISK]       ; verify that a hard disk is attached
        .PUSHJ  @[FMTARG]       ; get the partition and ask for confirmation
        CDF     1               ; the IDE disk data lives in field one
        DCA     @[DKPART]       ; save the partition number
        CDF     0               ; ...
        TAD     ZM256           ; the record size for IDE disk is 256 words
        DCA     RECSIZ          ; ...

; Do pass 1...
        DCA     FMTCNT          ; an IDE partition always holds 4096 blocks
        TAD     [DISKWR]        ; point to the correct I/O routine
        DCA     @[FMTWRP]       ; and point pass 1 towards that
        .PUSHJ  @[FMTP1]        ; go do pass 1

; And do pass 2...
        DCA     FMTCNT          ; reset the block count to 4096
        TAD     [DISKRD]        ; and point pass 2 to the disk read routine
        DCA     FMTRDP          ; ...
        .PUSHJ  FMTP2           ; and away we go!

; We've tested the entire disk...
FRDONE: JMS     @ZINLMES        ; let the operator know we're done
         FM3MSG                 ; "Finished"
        JMP     @ZCRLF          ; finish the line and we're done
        .TITLE  RF Command - Format a RAM Disk


;   The RF command will "format" a RAM disk virtual drive and it's essentially
; identical to the DF command that formats IDE disks.
;
;       >RF u           - format RAM disk unit u
;
```

```
RFRMAT: .PUSHJ  @[FMTARG]       ; get the unit and ask for confirmation
        CDF     1               ; the RAM disk data lives in field one
        DCA     @[RDUNIT]       ; save the unit
        CDF     0               ; ...
        JMS     @ZPUSHJ1        ; (cross field call)
         RAMSEL                 ; try to select this RAM disk unit
        SZL CLA                 ; was the unit number legal ??
         JMP    @[NOUNIT]       ; nope - quit while we're ahead!
        TAD     ZM128           ; the record size for RAM disk is 128 words
        DCA     RECSIZ          ; ...

; Do pass 1...
        CDF     1               ; get the size of this RAM disk unit
        TAD     @[RAMUSZ]       ; which is left here by RAMSEL
        DCA     FMTCNT          ; save it for pass 1
        CDF     0               ; ...
        TAD     [RAMDWR]        ; get the correct I/O routine
        DCA     @[FMTWRP]       ; and point pass 1 towards that
        .PUSHJ  @[FMTP1]        ; go do pass 1

; And do pass 2...
        CDF     1               ; get the size of this RAM disk unit
        TAD     @[RAMUSZ]       ; which is left here by RAMSEL
        DCA     FMTCNT          ; save it for pass1
        CDF     0               ; ...
        TAD     [RAMDRD]        ; and point pass 2 to the disk read routine
        DCA     FMTRDP          ; ...
        .PUSHJ  FMTP2           ; and away we go!

; We've tested the entire disk...
        JMP     FRDONE          ; say "Finished" and we're done!

        .PAGE
        .TITLE  LP Command - Load Binary Paper Tapes from the Console


;   The LP command loads a "paper tape" in standard PDP-8 BIN loader format
; from the console.  If the console is actually an ASR-33 and you actually
; have a real PDP-8 paper tape, then this will probably even work, but a more
; likely situation is that you're using a PC with a terminal emulator.  In
; that case the paper tape image can be downloaded from the PC's disk.
;
;   The loader accepts all standard BIN data frames, including field changes,
; and correctly calculates and verifies the tape checksum.  If the checksum
; matches then the number of words loaded is printed - otherwise a checksum
; error message is generated.  When initially started, this routine ignores
; all input until two consectutive leader codes (octal 200) are found - this
; allows us to ignore any extra ASCII characters from the terminal emulator
; (such as carriage returns, spaces, etc).
;
;   Since we're using the real console, the same one that you're typing
; commands on, for input we have a problem in that we need some way to
; terminate loading.  Control-C won't work since the BIN loader eats all
; eight bit characters.  A hardware reset isn't a good idea, since the POST
; memory test will erase everything we've loaded.  Instead we use a special
; routine, CONGET, to read characters from the console and this routine has a
; timeout built in.  If we go approximately 5 seconds without any input then
; the loader is terminated.
CONLOD: .PUSHJ  @ZEOLNXT        ; this command has no operands
        DCA     PNLMEM          ; files are always lodaded into main memory

; Look for two consecutive bytes of leader code...
BINLO1: TAD     [-2]            ; we need two bytes of leader/trailer
        DCA     COUNT           ; ...
BINLO2: .PUSHJ  CONGET          ; go read a byte of input
        TAD     [-200]          ; is this a leader code ??
        SZA CLA                 ; ??
         JMP     BINLO1         ; no -- keep looking for two
        ISZ     COUNT           ; yes -- is this the second in a row ??
         JMP     BINLO2         ; no -- go look for the next one

; Here after we have 2 bytes of leader -- look for the end of the leader...
```

```
BINLO3: .PUSHJ  CONGET          ; get another byte of data
        TAD     [-200]          ; are we still in the leader ??
        SNA                     ; ???
         JMP     BINLO3         ; yes -- keep looking
        TAD     [200]           ; no -- restore the character
        DCA     WORD            ; and remember it for later

; Now actually start loading data...
        DCA     CHKSUM          ; start with a zero checksum
        TAD     [200]           ; set the default load address to location 200
        DCA     ADDR            ; ...
        DCA     ADRFLD          ; in field zero

; Decode the type of the next load record...
BINLO5: CAM                     ; ...
        TAD     WORD            ; Get the last character we read
        AND     [200]           ; Is this a single byte frame ???
        SZA CLA                 ; ??
         JMP     BINLO7         ; Yes -- this is EOF or a field setting

; Load a two frame record (either data or an address)...
        TAD     WORD            ; get the first byte back again
        DCA     BINCH1          ; and remember that
        .PUSHJ  CONGET          ; then go read the next byte
        DCA     BINCH2          ; and save that
        TAD     BINCH1          ; get the first byte
        AND     [77]            ; trim it to just 6 bits
        BSW                     ; put it in the left half
        MQL                     ; and save it in the MQ for now
        TAD     BINCH2          ; then get the second character
        AND     [77]            ; trim it to 6 bits too
        MQA                     ; and OR it with the first character
        DCA     VALUE           ; remember what we read

; Determine what to do with this word...
        .PUSHJ  CONGET          ; look ahead one byte
        DCA     WORD            ; save that character
        TAD     WORD            ; and get it back
        TAD     [-200]          ; is this the end of the tape ??
        SNA CLA                 ; ??
         JMP     BINLO8         ; yes -- we read a checksum word
        TAD     CHKSUM          ; no -- checksum the two characters we read
        TAD     BINCH1          ; ...
        TAD     BINCH2          ; ...
        DCA     CHKSUM          ; ...
        TAD     BINCH1          ; then look at the first character
        AND     [100]           ; is this an address or data frame ??
        SZA CLA                 ; skip if it's data
         JMP     BINLO6         ; no -- it is an address

; Load this word of data into memory...
        TAD     VALUE           ; get the word back
        .PUSHJ  @ZDANDV         ; and write it into memory
        ISZ     ADDR            ; automatically advance the address
         NOP                    ;  (and ignore any wrap around)
        JMP      BINLO5         ; then go process the next frame

; This word is an address...
BINLO6: TAD     VALUE           ; get the 12 bits of data
        DCA     ADDR            ; and change to that address
        JMP     BINLO5          ; then go process the next frame

; Here of the current frame is a field setting...
BINLO7: TAD     WORD            ; get the last character back again
        AND     [100]           ; see if it is really a field frame
        SNA CLA                 ; ???
         JMP     BINLO8         ; no -- treat it like a trailer code
        TAD     WORD            ; get the field back
        AND     ZK70            ; we only want these bits
        DCA     ADRFLD          ; and change to the selected field
        .PUSHJ  CONGET          ; Then look ahead one byte
        DCA     WORD            ; ...
```

```
        JMP     BINLO5          ; and go process that frame

; Here when we find the checksum byte...
BINLO8: TAD     VALUE           ; get the checksum byte
        CIA                     ; make it negative
        TAD     CHKSUM          ; and add it to our checksum
        DCA     CHKSUM          ; this should leave zero
        .PUSHJ  @[TCKSUM]       ; go type the checksum and return
        JMP     BINLO1

; Temporary storage for BIN loader routine...
BINCH1: .BLOCK  1               ; the first of a two character frame
BINCH2: .BLOCK  1               ; the second of a two character frame
        .TITLE  Paper Tape Console Input Routine


;   This routine will read a character from the console, waiting if there is
; none ready right now, and with a timeout if one doesn't arrive soon.  It is
; intended to be used only with the paper tape binary loader routine, and most
; "textual" input should be done via the INCHRS or INCHWL routines.  Since the
; user cannot type control-C to abort the paper tape loader (data is being read
; from the console, remember ?) this routine provides a timeout feature to
; prevent the monitor from becoming 'hung'.  If no character is received from
; the console in approximately 10 seconds, a control-C is simulated by jumping
; to RESTA.
FTCONT=200.     ; approximately 10 seconds with a 4.9152Mhz clock

CONGET: CLA                     ; ...
        TAD     [-FTCONT]       ; get the console timeout time
        DCA     IRMA            ; and set up a counter

; Try to read a character...
CONGE1: KSF                     ; is there a character there ???
         JMP     CONGE2         ; no -- check the timer
        CLA                     ; yes -- clear the timer
        KRB                     ; and get the character
        .POPJ                   ; then return that

;   Here to keep the timeout counter. The loop between CONGE1 and CONGE2
; requires 56 states, or approximately .1835 seconds at 2.5Mhz.  This is
; executed FTCONT times for the overall timeout.
CONGE2: IAC                     ; increment the timer
        SZA                     ; has it counted to 4096 ???
         JMP     CONGE1         ; no -- keep waiting
        ISZ     IRMA            ; yes -- have we waited long enough ??
         JMP     CONGE1         ; no -- wait a little longer
        NL0003                  ; yes -- simulate a control-C
        .PUSHJ  @ZOUTCHR        ; echo ^C
        JMP     @ZRESTA         ; and restart

        .PAGE
        .TITLE  RD and DD Commands - Dump Disk (RAM and IDE) Records


;   These commands dump one or more disk records, in octal, to the console.
; What you get from DP is exactly how the OS/8 device driver sees the disk
; data.  Each command accepts one, two or three parameters.  The first is unit
; number for RAM Disk (RD) commands, or the partition number for IDE Disk (DD)
; commands.  The second parameter is the number of the block to be dumped, in
; octal.  If this number is omitted then the ENTIRE disk will be dumped which,
; although legal, will take quite a while!  The third parameter is the count
; of pages (for RAM Disk) or blocks (for IDE disk) to be dumped and, if
; omitted, this defaults to 1. For example:
;
;       >RD 0 0         - dump only page 0 (the boot block) of RAM disk unit 0
;       >DD 2 100       - dump only page 100 (octal) of IDE partition 2
;       >RD 1 100 77    - dump 64 (77 octal) pages of unit 1 from 100 to 177
;       >DD 0           - dump ALL of IDE partition zero (4095 blocks!)
;

; Enter here for the RD command...
RDDUMP: TAD     [RAMDRD]        ; point to the RAM disk read routine
```

```
        DCA     RDPTR           ; modify the code to use that
        TAD     ZM128           ; get the record size for RAM disk
        DCA     RECSIZ          ; and save that
        JMP     PARSDX          ; fall into the regular code now

; And here for the DD command...
DDDUMP: .PUSHJ  @[NODISK]       ; verify that a hard disk exists
        TAD     [DISKRD]        ; point to the IDE disk read routine
        DCA     RDPTR           ; and use that instead
        TAD     ZM256           ; IDE disk uses 256 word records
        DCA     RECSIZ          ; ...

; Parse the argument lists for either command...
PARSDX: .PUSHJ  @ZOCTNW         ; read the unit/partition number (required)
        CDF     1               ; all disk data lives in field 1
        TAD     WORD            ; get what we found
        DCA     @[DKPART]       ; save both the partition number
        TAD     WORD            ; ...
        DCA     @[RDUNIT]       ; and the unit number
        DCA     @ZDKRBN         ; set the default starting block/page to zero
        DCA     @ZRDPAGE        ; ...
        CDF     0               ; back to the current field
        DCA     RECCNT          ; make the default record count the whole disk

; See if there's a starting page number on the command line...
        .PUSHJ  @ZSPACM0        ; are there any more characters in the command?
        SNA CLA                 ; skip if there are
         JMP     DDUMP1         ; nope - start dumping now
        .PUSHJ  @ZBACKUP        ; yes - re-read the character
        .PUSHJ  @ZOCTNW         ; and read the page/block number
        CDF     1               ; back to field 1
        TAD     WORD            ; get the starting block/page number
        DCA     @ZDKRBN         ; and save it for both RAM disk and IDE disk
        TAD     WORD            ; ...
        DCA     @ZRDPAGE        ; ...
        CDF     0               ; back to our field
        NLM1                    ; now the default record count is one
        DCA     RECCNT          ; ...

; See if there's a page/block count too..
        .PUSHJ  @ZSPACM0        ; still more characters?
        SNA CLA                 ; skip if there are
         JMP     DDUMP1         ; nope - start dumping now
        .PUSHJ  @ZBACKUP        ; yes - re-read the character
        .PUSHJ  @ZOCTNW         ; and read the page count
        TAD     WORD            ; ...
        CIA                     ; make it negative for ISZ
        DCA     RECCNT          ; and save the count
        .PUSHJ  @ZEOLTST        ; finally, this has to be the end of the line

; Read a page from the disk into the panel memory buffer and dump it...
DDUMP1: JMS     @ZPUSHJ1        ; (call field 1 routine)
         PNLBUF                 ; set the disk buffer to DSKBUF
        TAD     RECSIZ          ; pass the record size to the I/O routine
        JMS     @ZPUSHJ1        ; (cross field call)
RDPTR:   .BLOCK  1              ; gets overwritten with DISKRD or RAMDRD!
        SZL                     ; were there any errors detected ?
         JMP     DIOERR         ; yes - report it and quit
        TAD     RECSIZ          ; nope - get the size of this record
        .PUSHJ  @[DDBUF]        ; and go dump the DSKBUF
        CDF     1               ; disk data lives in field 1
        ISZ     @ZDKRBN         ; increment both the IDE block
        ISZ     @ZRDPAGE        ; and RAM disk page
        CDF     0               ; ...
        ISZ     RECCNT          ; have we done all we need to?
         JMP     DDUMP1         ; nope - go dump another one
        .POPJ                   ; yes - we're done (finally!!)

; Here if a disk I/O error occurs...
DIOERR: CDF     0               ; just in case
        DCA     VALUE           ; save the error code for a minute
        JMS     @ZINLMES        ; say
```

```
         ERRDIO                   ;  "?I/O Error "
        TAD     VALUE           ; get the error status
        .PUSHJ  @ZTOCT4C        ; type it and a CRLF
        JMP     @ZRESTA         ; and abort this command completely
        .TITLE  RL and DL Commands - Load Disk (RAM and IDE) Records


;   The DL and RL commands allow a disk to be downloaded over the console
; serial port.  The format of the data expected is identical that that
; generated by the RD and DD (dump RAM/IDE disk) commands, which makes it
; possible to upload a disk image to the PC and then later download the same
; image back to the SBC6120.  Since all the data is simple printing ASCII
; text, any terminal emulator program can be used to capture and replay the
; data will suffice.
;
;       >RL u    - download data to RAM disk unit u
;       >DL pppp - download data to IDE disk partition pppp

; Enter here for the RL command...
RLLOAD: TAD     [RAMDWR]        ; point to the RAM disk write routine
        DCA     WRPTR           ; modify the code to use that
        TAD     ZM128           ; set the record (page) size for RAM disk
        DCA     RECSIZ          ; ...
        JMP     DLOAD           ; fall into the regular code

; Enter here for the DL command...
DLLOAD: .PUSHJ  @[NODISK]       ; verify that a hard disk is attached
        TAD     [DISKWR]        ; this time use the IDE disk write routine
        DCA     WRPTR           ; ...
        TAD     ZM256           ; and the record (block) size is 256
        DCA     RECSIZ          ; ...

; Parse the argument for the DL and RL commands...
DLOAD:  .PUSHJ  @ZOCTNW         ; read the unit/partition number (required)
        .PUSHJ  @ZEOLTST        ; that has to be followed by the end of line
        CDF     1               ; all disk data lives in field 1
        TAD     WORD            ; get the number entered
        DCA     @[RDUNIT]       ; and set the RAM disk unit number
        TAD     WORD            ; ...
        DCA     @[DKPART]       ; and the IDE disk partition number
        CDF     0               ; back to our field now

; Here to read another disk page of data from the host...
DLOAD1: TAD     RECSIZ          ; pass the block size in the AC
        .PUSHJ  @[LDBUF]        ; load the disk buffer from the serial port
        CDF     1               ; (disk data lives in field 1)
        DCA     @ZDKRBN         ; save the address of the block we read
        TAD     @ZDKRBN         ; ...
        DCA     @ZRDPAGE        ; and the page number too
        CDF     0               ; (back to our field now)
        JMS     @ZPUSHJ1        ; (call field 1 routine)
         PNLBUF                 ; set the disk buffer to DSKBUF
        TAD     RECSIZ          ; pass the record size to the I/O routine
        JMS     @ZPUSHJ1        ; (call a field 1 routine)
WRPTR:   .BLOCK 1               ; gets modified to either DISKWR or RAMDWR
        SZL                     ; were there any I/O errors?
         JMP     DIOERR         ; yes - go report that and quit
        JMP     DLOAD1          ; go read another page

        .PAGE
        .TITLE  Dump Disk Buffer on Console


;   This routine will dump the contents of DSKBUF on the console in ASCII.
; For each block dumped the output format consists of 33 lines of data, where
; the first 32 lines contain a disk address in the format <block> "." <offset>
; (e.g. "0122.0160" is word 160 (octal) of block 122 (octal)) followed by 8
; words of data, also in octal.  The 33rd line contains just a single octal
; number, a checksum of all 256 words in the block.
;
;   This format is exactly the same input that's accepted by the LDBUF, which
; allows you to capture the output of a disk dump on a PC terminal emulator
```

```
; and then download the same data later to a different disk.  This is the
; primary motivation for the checksum - it isn't too useful to humans, but
; it will guard against errors in the upload/download procedure.
;
;   This routine should be called with the number of words to dump in the
; AC, which will normally be either -256 (to dump an IDE block) or -128
; (for a RAM disk page).
DDBUF:  DCA     DDCNT           ; save the count of words to dump
        DCA     COUNT           ; and clear the current word count
        TAD     [DSKBUF-1]      ; set up X1 to point to the buffer
        DCA     X1              ; ...
        DCA     CHKSUM          ; and clear the checksum

; Start a new line of data...
DDBUF2: CDF     1               ; ...
        TAD     @ZDKRBN         ; get the page/block number we're dumping
        CDF     0               ; ...
        .PUSHJ  @ZTOCT4         ; type it in octal
        .PUSHJ  @[TDOT]         ; then type the separator
        TAD     COUNT           ; then type the offset
        .PUSHJ  @[TOCT3]        ; ...
        .PUSHJ  @[TSLASH]       ; another separator character
        .PUSHJ  @ZTSPACE        ; ...

; Dump eight words of data, in octal...
DDBUF3: CDF     1               ; the disk buffer is in field 1
        TAD     @X1             ; get another word
        CDF     0               ; and go back to our field
        DCA     VALUE           ; save it for a minute
        TAD     VALUE           ; get it back
        TAD     CHKSUM          ; and accumulate a checksum
        DCA     CHKSUM          ; ...
        TAD     VALUE           ; now we're ready to type the data
        .PUSHJ  @ZTOCT4S        ; in octal, with a space
        ISZ     COUNT           ; count the number we've done
        TAD     COUNT           ; ...
        AND     ZK7             ; have we done a complete row of eight?
        SZA CLA                 ; ???
         JMP     DDBUF3         ; no - keep going

; Here after we've finished a line of eight data words...
        .PUSHJ  @ZCRLF          ; start a new line
        TAD     COUNT           ; see if we've done the whole page/block
        TAD     DDCNT           ; compare to the block size
        SZA CLA                 ; ???
         JMP     DDBUF2         ; not there yet - keep dumping
        TAD     CHKSUM          ; type just the checksum
        JMP     @ZTOCT4C        ; in octal, with a CRLF, and we're done


; Local storage for DDBUF...
DDCNT:  .BLOCK  1               ; the number of words in this buffer
        .TITLE  Load Disk Buffer from Console


;   This routine loads the disk buffer with data from a disk block image
; transmitted over the console port.  The format of the data expected is
; identical that that generated by the DDBUF routine, which makes it possible
; to upload a disk image to the PC and then later download the same image back
; to the SBC6120 with DL.  Since all the data is simple printing ASCII text,
; any terminal emulator program can be used to capture and replay the data.
;
;   LDBUF prompts for each line of data with a ":", and most terminal emulator
; programs for the PC can be set to look for this prompting character before
; transmitting the next line.  This eliminates the need to insert fixed delays
; to avoid overrunning the SBC6120.  Since LDBUF reads only printing ASCII
; characters, a download can be aborted at any time just by typing a control-C
; and there's no need for a timeout the way there is with loading paper tape
; images.
;
;   The expected block size, either -128. for RAM disk or -256. for IDE disk,
; should be passed in the AC.  The data read is left in DSKBUF, and the
```

```
; page/block number, extracted from the data, is left in LDPAGE.  Note that
; in the event a checksum or syntax error is found, LDBUF prints an error
; message and restarts the command scanner.  In that case control never
; returns to the caller!
LDBUF:  DCA     DDCNT           ; save the expected block size
        TAD     [DSKBUF-1]      ; initialize X1 to point at our buffer
        DCA     X1              ; ...
        DCA     COUNT           ; count the data words read here
        STA                     ; the current disk page is unknown
        DCA     LDPAGE          ; ...
        DCA     CHKSUM          ; clear the checksum accumulator

; Read the next line of data...
LDBUF2: TAD     [":"]           ; prompt for data with a ":"
        .PUSHJ  @[INCHWL]       ; and read an entire line of data
        TAD     COUNT           ; see how many words we've read so far
        TAD     DDCNT           ; is it time for the checksum?
        SNA CLA                 ; ???
         JMP    LDBUF5          ; yes - go parse a checksum record

;   First parse the disk page number and offset.  The offset has to match the
; number of words we've already read, but the disk address is slightly more
; complicated.  For the first data record in a page we allow the address to
; be anything, and that tells us which disk page is to be written.  Each data
; record after that up to the end of the page has to have the same disk
; address as the first one.  This allows disk pages to be loaded in any random
; order and, more importantly, it allows unused pages to be skipped.
        .PUSHJ  @ZOCTNW         ; go read an octal number
        TAD     WORD            ; get the value we scanned
        CIA                     ; compare it to LDPAGE
        TAD     LDPAGE          ; ???
        SNA CLA                 ; do they match ?
         JMP    LDBUF3          ; yes - all is well
        ISZ     LDPAGE          ; no - is this the first data record?
         JMP    @ZCOMERR        ; not that either - the data is corrupt
        TAD     WORD            ; yes - just use this page number without
        DCA     LDPAGE          ;  ... question
LDBUF3: TAD     SAVCHR          ; get the separator character
        TAD     [-"."]          ; it has to be a "."
        SZA CLA                 ; ???
         JMP    @ZCOMERR        ; nope - bad load format
        .PUSHJ  @ZOCTNW         ; now read the relative offset within the page
        TAD     WORD            ; ...
        CIA                     ; it has to match our data count
        TAD     COUNT           ; does it?
        SZA CLA                 ; ???
         JMP    @ZCOMERR        ; nope - more bad data
        TAD     SAVCHR          ; one last test
        TAD     [-"/"]          ; the separator this time has to be a slash
        SZA CLA                 ; ???
         JMP    @ZCOMERR        ; another corrupted data record

;   Now read the rest of the data record, which should consist of exactly
; eight data words, in octal...
LDBUF4: .PUSHJ  @ZOCTNW         ; scan the next data word
        TAD     WORD            ; get the value we read
        CDF     1               ; remember that the disk buffer is in field 1
        DCA     @X1             ; and store the data word
        TAD     WORD            ; accumulate a checksum of the data words
        TAD     CHKSUM          ; ... we read
        DCA     CHKSUM          ; ...
        CDF     0               ; back to home ground
        ISZ     COUNT           ; count the number of words we've read
        TAD     COUNT           ; let's have a look at it
        AND     ZK7             ; have we read exactly eight words?
        SZA CLA                 ; skip if we have
         JMP    LDBUF4          ; no - go read another data word
        .PUSHJ  @ZGET           ; yes - after eight data words
        SZA CLA                 ; ... the next thing should be the EOL
         JMP    @ZCOMERR        ; not EOL - this data is corrupted somehow
        JMP     LDBUF2          ; this is the EOL - go read another record
```

```
;   We get here when we're read 128 or 256 words of data - the next thing we
; expect to find is a checksum record, which is a single octal number all by
; itself.  This has to match the checksum we've calculated or the data is
; corrupted.
LDBUF5: .PUSHJ  @ZOCTNW         ; scan an octal value
        TAD     WORD            ; and get what we found
        CIA                     ; ...
        TAD     CHKSUM          ; compare it to the checksum we accumulated
        SZA CLA                 ; they have to be the same!
         JMP     DERCKS         ; they aren't - bad checksum for data
        TAD     SAVCHR          ; get the next character
        SZA CLA                 ; it has to be the EOL
         JMP     @ZCOMERR       ; no - the syntax of this record is wrong

; The checksum matches - all is well!
        TAD     LDPAGE          ; return the page number in the AC
        .POPJ                   ; ...

; Here if the data checksum doesn't match...
DERCKS: JMS     @ZERROR         ; print a message and restart
         ERRCKS                 ; ?DATA CHECKSUM MISMATCH

; Local storage for LDBUF...
LDPAGE: .BLOCK  1               ; page number being read

        .PAGE
        .TITLE  PC Command - Copy an IDE Disk Partition


;   The PC command will copy an entire disk partition to another partition.
; It's a convenient way to create backups of OS/8 partitions, especially since
; most modern IDE drives have room for thousands of OS/8 partitions!
;
;       >PC ssss dddd   - copy IDE partition ssss to partition dddd
;
PCOPY:  .PUSHJ  @ZOCTNW         ; read the source partition number
        TAD     WORD            ; ...
        DCA     CPYSRC          ; ...
        .PUSHJ  @[SPATST]       ; the next character has to be a space
        .PUSHJ  @ZOCTNW         ; then read the destination partition
        TAD     WORD            ; ...
        DCA     CPYDST          ; ...
        .PUSHJ  @ZEOLTST        ; that'd better be all there is

; Ask for confirmation before overwriting the destination partition...
        JMS     @ZINLMES        ; say
         CCFMSG                 ;  "Overwrite partition/unit "
        TAD     CPYDST          ; and then type the partition number
        .PUSHJ  @ZTOCT4S        ; ...
        .PUSHJ  @[CONFRM]       ; then wait for a "Y" or "N"
        SNL CLA                 ; did he answer yes???
         JMP     @ZRESTA        ; nope - just quit now

; Prepare to begin copying...
        JMS     @ZINLMES        ; say
         CP1MSG                 ;  ... "Copying "
        CDF     1               ; reset the current block number
        DCA     @ZDKRBN         ; ...
        CDF     0               ; ...

; Read the next block from the SRC partition...
PCOPY1: JMS     @ZPUSHJ1        ; (cross field call)
         PNLBUF                 ; setup a temporary disk buffer in panel memory
        TAD     CPYSRC          ; get the source partition
        CDF     1               ; ...
        DCA     @[DKPART]       ; and point DISKRD there
        CDF     0               ; ...
        TAD     ZM256           ; the IDE record size is always 256 words
        JMS     @ZPUSHJ1        ; (cross field call)
         DISKRD                 ; and go read a block from the disk
        SZL                     ; disk error ??
         JMP     @[DIOERR]      ; yes - go report it and give up
```

```
; And now write it to the destination...
PCOPY2: JMS     @ZPUSHJ1        ; (cross field call)
         PNLBUF                 ; setup the panel memory disk buffer
        TAD     CPYDST          ; change DKPART to the destination partition
        CDF     1               ; ...
        DCA     @[DKPART]       ; ...
        CDF     0               ; ...
        TAD     ZM256           ; load the record size for DISKWR
        JMS     @ZPUSHJ1        ; (cross field call)
         DISKWR                 ; and go write a block to the disk
        SZL                     ; any disk errors?
         JMP     @[DIOERR]      ; yes - give up

; Print a dot every so often to make a simple "progress bar"...
PCOPY3: CDF     1               ; ...
        ISZ     @[DKRBN]        ; increment the block number
         SKP                    ; still more to go
          JMP     PCOPY4        ; we've done all 4096 blocks!
        TAD     ZM256           ; the format command uses the record size as
        CMA                     ;  a mask for printing the progress bar
        AND     @[DKRBN]        ;  and so we will too
        CDF     0               ; ...
        SZA CLA                 ; time for another dot??
         JMP     PCOPY1         ; nope - just keep copying
        .PUSHJ  @[TDOT]         ; print a dot to show our progress
        JMP     PCOPY1          ; and another page or block

; All done...
PCOPY4: CDF     0               ; ...
        JMS     @ZINLMES        ; say
         CP2MSG                 ; " Done"
        JMP     @ZCRLF          ; and that's all!

        .PAGE
        .TITLE  Free Space for Future Expansion!

        .PAGE   31
        .TITLE  Type ASCII Strings


;   This routine will type a ASCIZ string stored in field 1 using the standard
; OS/8 "3 for 2" packing system.  This format is used by the monitor to store
; help and error messages.  On call, the address of the string is passed in the
; AC and, on return, the AC will always be cleared.
OUTSTR: TAD     [-1]            ; auto index registers pre-increment
        DCA     X1              ; ...
        NLM3                    ; load the AC with -3
        DCA     DIGITS          ; and initialize the character counter

; Get the next character and output it...
OUTST1: CDF     1               ; strings always live in field 1
        ISZ     DIGITS          ; which character are we on?
         JMP     OUTST2         ; first or second - they're easy

; Extract the third character from a triplet...
        NLM3                    ; re-initialize the character counter
        DCA     DIGITS          ; ...
        NLM2                    ; then load the AC with -2
        TAD     X1              ; and backup the string pointer
        DCA     X1              ; ...
        TAD     @X1             ; get the first word of the pair
        AND     ZK7400          ; get the upper four bits of the word
        BSW                     ; position them in the upper bits of the byte
        CLL RTL                 ; ...
        MQL                     ; save it in the MQ for a while
        TAD     @X1             ; then get the second word again
        AND     ZK7400          ; the upper four bits again
        BSW                     ; become the lower for bits of the byte
        CLL RTR                 ; ...
        MQA                     ; put the byte together
        JMP     OUTST3          ; and type it normally
```

```
; Here for the first or second character of a triplet...
OUTST2: TAD     @X1             ; get the character
OUTST3: AND     ZK177           ; trim it to just seven bits
        CDF     0               ; restore the original data field
        SNA                     ; end of string ?
         .POPJ                  ; yes - we can quit now
        .PUSHJ  @ZOUTCHR        ; nope - type this one too
        JMP     OUTST1          ; and go do the next

;   This routine does exactly the same thing as OUTSTR, except that it allows
; the message pointer to be passed in line, via a JMS instruction
INLMES: 0                       ; call here via a JMS instruction
        CLA                     ; ...
        TAD     @INLMES         ; fetch the string address
        .PUSHJ  OUTSTR          ; type it out
        ISZ     INLMES          ; skip over the address
        JMP     @INLMES         ; and return

;   This routine will type an ASCIZ string, packed one character per word and
; terminated with a null character, on the terminal.  The address of the
; string, -1, should be loaded into the AC before calling this routine, and
; the AC will always be cleared on return.
TASCIZ: DCA     X1              ; save the pointer to the string
TASCI1: TAD     @X1             ; and get the first character
        SNA                     ; is this the end of the string ??
         .POPJ                  ; yes -- quit now
        .PUSHJ  @ZOUTCHR        ; no -- type this character
        JMP     TASCI1          ; and then loop until the end

;   This routine is identical to TASCIZ, except that the string is stored in
; field 1, rather than field 0...
TASZF1: DCA     X1              ; save the pointer to the string
        CDF     1               ; the string is in field 1
        TAD     @X1             ; and get the next character
        CDF     0               ; back to our field
        SNA                     ; is this the end of the string ??
         .POPJ                  ; yes -- quit now
        .PUSHJ  @ZOUTCHR        ; no -- type this character
        JMP     TASZF1+1        ; and then loop until the end
        .TITLE  Type SIXBIT Words, and Characters


;   This routine will type the two character SIXBIT word contained in the AC.
; It always types exactly two characters, so if the second character is a null
; (00), then a trailing blank appears.  The AC is cleared on return.
TSIXW:  DCA     WORD            ; Save the 2 characters
        TAD     WORD            ; And get them back
        BSW                     ; Position the first one
        .PUSHJ  TSIXC           ; And type it out
        TAD     WORD            ; No -- get the second character
                                ; And fall into the TSIXC routine

;   This routine will type a single SIXBIT character from the right
; byte of the AC.  The AC will be cleared on return.
TSIXC:  AND     [77]            ; Trim the character to just 6 bits
        TAD     [" "]           ; No -- convert the character to ASCII
        JMP     @[THCHAR]       ; And type it out
        .TITLE  Type Decimal Numbers


;   This routine will type the contents of the AC in decimal.  It always
; treats the AC as an unsigned quantity and will type numbers from 0 to 4095.
; It uses locations WORD and COUNT and the AC is always cleared on return.
TDECNW: DCA     WORD            ; remember the number to be typed
        DCA     DIGITS          ; and clear the quotient
        TAD     WORD            ; get the dividend back again

;   Divide by 10 via repeated subtraction...
TDECN1: CLL                     ; make sure the LINK is clear
        TAD     [-10.]          ; subtract 10 from the dividend
        SNL                     ; did it fit ???
```

```
         JMP     TDECN2         ; no -- go get the remainder
        ISZ     DIGITS          ; yes -- increment the quotient
        JMP     TDECN1          ; and keep dividing

;   Now figure the remainder...
TDECN2: TAD     [10.]           ; correct the remainder
        .PUSH                   ; and save it on the stack
        CLA                     ; get the quotient
        TAD     DIGITS          ; ...
        SNA                     ; is it zero ???
         JMP     TDECN3         ; yes -- proceed
        .PUSHJ  TDECNW          ; no type that part first

;   Here to type the digit and return...
TDECN3: .POP                    ; restore the remainder
        JMP     @[TDIGIT]       ; type it in ASCII and return
        .TITLE  Type Octal Numbers


;   This routine will type a 4 digit octal number passed in the AC.   It always
; prints exactly four digits, with leading zeros added as necessary.  The AC
; will be cleared on return.
TOCT4:  DCA     WORD            ; save the number to type
        TAD     [-4]            ; and get the number of iterations
TOCTN:  DCA     DIGITS          ; ...

; Extract one digit and print it...
TOCTL:  TAD     WORD            ; get the remaining bits
        CLL RTL                 ; shift them left 2 bits
        RTL                     ; and then 2 more (remember the link!)
        DCA     SAVCHR          ; remember that for a later
        TAD     SAVCHR          ; and we also need it now
        RAR                     ; restore the extra bit (in the link)
        DCA     WORD            ; then remember the remaining bits
        TAD     SAVCHR          ; get the digit back
        AND     ZK7             ; trim it to just 3 bits
        .PUSHJ  @[TDIGIT]       ; type it out

; Here after we have typed another digit...
        ISZ     DIGITS          ; is this enough ??
         JMP     TOCTL          ; no -- keep typing
        .POPJ                   ; yes -- quit now

;   This routine is identical to TOCT4, except that it types only three digits
; with leading zeros.  It's useful for printing eight bit quantities...
TOCT3:  R3L                     ; throw away the most significant digit
        DCA     WORD            ; save the value to be typed
        NLM3                    ; get the number of iterations
        JMP     TOCTN           ; and join the regular code

; This small routine will type an octal number in the AC followed by a space.
TOCT4S: .PUSHJ  TOCT4           ; then type the data in octal
        JMP     @ZTSPACE        ; finally type a space and return

; This small routine will type an octal number from the AC followed by a CRLF.
TOCT4C: .PUSHJ  TOCT4           ; and type that in octal
        JMP     @ZCRLF          ; finish with a CRLF

        .PAGE
        .TITLE  Type 15 Bit Addresses


;   This routine will type a 15 bit address, passed in location ADDR, and with
; the field is in location ADRFLD.  The address will be typed as a 5 digit
; octal number, and then followed by a "/" character and a space. The initial
; contents of the AC are ignored and the AC is always cleared on return.
TADDR:  CLA                     ; ...
        TAD     ADRFLD          ; get the high 3 bits of the address
        .PUSHJ  @[TFIELD]       ; type that out
        TAD     ADDR            ; then get the address
        .PUSHJ  @ZTOCT4         ; and type all 12 bits of that
        .PUSHJ  @[TSLASH]       ; type a slash as a separator
```

```
        JMP     @ZTSPACE        ; finish with a space

;   This routine will type a single octal digit which represents a memory
; field.  The field should be passed in the AC.
TFIELD: RAR                     ; right justify the field number
        RTR                     ; ...
        AND     ZK7             ; trim it to just 3 bits
        JMP     @[TDIGIT]       ; and fall into TDIGIT...
        .TITLE  Scan Addresses


; This routine will read a 15 bit address into registers ADDR and ADRFLD.
RDADDR: .PUSHJ  @[OCTNF]        ; read the 15 bit address
        TAD     WORD            ; get the low order bits
        DCA     ADDR            ; and put them in ADDR
        .POPJ                   ; that's it...

; This routine will read a 15 bit address into registers HIGH and HGHFLD.
RDHIGH: .PUSHJ  RDADDR          ; read a 15 bit address
        TAD     ADDR            ; get the low order bits
        DCA     HIGH            ; into HIGH
        TAD     ADRFLD          ; get the field
        DCA     HGHFLD          ; into HGHFLD
        .POPJ                   ; and that's all

; This routine will read a 15 bit address into registers LOW AND LOWFLD.
RDLOW:  .PUSHJ  RDADDR          ; the same thing as before
        TAD     ADDR            ; ...
        DCA     LOW             ; only the names have changed
        TAD     ADRFLD          ; ...
        DCA     LOWFLD          ; ...
        .POPJ                   ; ...
        .TITLE  Scan an Address Range


;   This routine will read either one or two octal numbers which describe a
; range of memory addresses.  A range may be a single number (in which case
; the starting and ending values are the same) or two numbers with a space
; character between them (in which case the first number is the starting value
; and the last is the ending value).  The starting value is always returned in
; locations LOW/LOWFLD and ADDR/ADRFLD, and the ending value is placed in
; HIGH/HGHFLD.  If two addresses were seen, the LINK will be set upon return;
; it is cleared if only one address was found.
RANGE:  .PUSHJ  RDLOW           ; first read the low part of the range
        .PUSHJ  @ZSPACM0        ; get the next non-space character
        TAD     [-"-"]          ; is it a range delimiter ??
        SZA CLA                 ; ???
         JMP     RANGE1         ; no -- this must be the single address type

;   Here for a two address range...
        .PUSHJ  RDHIGH          ; go read the high order part of the range
        TAD     LOW             ; make ADDR point to the starting point
        DCA     ADDR            ; ...
        TAD     LOWFLD          ; ...
        DCA     ADRFLD          ; ...
        .PUSHJ  TSTADR          ; then be sure the numbers are in order
        CML                     ; ...
        SZL CLA                 ; ???
         .POPJ                  ; yes -- return with the link set
        JMS     @ZERROR         ; no -- this isn't legal
         ERRRAN                 ; ?WRONG ORDER

;   Here for a single address range...
RANGE1: TAD     LOW             ; set the high equal to the low
        DCA     HIGH            ; ...
        TAD     LOWFLD          ; ...
        DCA     HGHFLD          ; ...
        CLL                     ; Then return with the link cleared
        .POPJ                   ; ...
        .TITLE  Address Arithmetic
```

```
;   This routine will increment the 15 bit address contained in registers
; ADDR and ADRFLD.  If the address increments past 77777, the link will be
; 1 on return; otherwise it is always 0.
NXTADR: CLA CLL                 ; ...
        ISZ     ADDR            ; increment the address
         .POPJ                  ; no wrap around -- leave the field alone
        TAD     ADRFLD          ; wrap around -- increment the field too
        TAD     [-70]           ; are we already in field 7 ??
        SPA                     ; ???
         CML                    ; no -- make the LINK be cleared on return
        TAD     [70+10]         ; restore and increment the field
        AND     ZK70            ; only allow these bits in the result
        DCA     ADRFLD          ; and put it back
        .POPJ                   ; ...

;   This routine will compare the 15 bit address in registers ADDR and
; ADRFLD to the address in registers HIGH and HGHFLD.  If ADDR/ADRFLD
; is less than HIGH/HGHFLD, the link will be zero on return.  If ADDR/ADRFLD
; is greater then or equal to HIGH/HGHFLD, the link will be one.
TSTADR: CLA CLL                 ; clear the AC and set L = 0
        TAD     ADRFLD          ; get the field
        CMA IAC CML             ; negate the field and set L = 1
        TAD     HGHFLD          ; compare to the high field
        SZA CLA                 ; are they equal ??
         .POPJ                  ; no -- the LINK has the correct status
        TAD     HIGH            ; yes -- compare the addresses
        CMA CIA                 ; L = 0 now
        TAD     ADDR            ; ...
        CLA                     ; clear the AC
        .POPJ                   ; but return the status in the LINK

;   This routine will swap the 15 bit address in ADDR/ADRFLD with the the
; 15 bit address in LOW/LOWFLD.  The AC is always cleared.
SWPADR: CLA                     ; ...
        TAD     LOW             ; get one value
        MQL                     ; and save it in the MQ
        TAD     ADDR            ; then get the other
        DCA     LOW             ; move it to the other place
        MQA                     ; and get the original one back
        DCA     ADDR            ; it goes in the second location
        TAD     LOWFLD          ; now do the same thing for fields
        MQL                     ; ...
        TAD     ADRFLD          ; ...
        DCA     LOWFLD          ; ...
        MQA                     ; ...
        DCA     ADRFLD          ; ...
        .POPJ                   ; that's all there is to it

        .PAGE
        .TITLE  Scan a Command Name


;   This routine will scan a command or register name for the monitor.  Names
; are always alphabetic, may not contain any digits, and are limited to one or
; two letters.  The result is stored, in SIXBIT, in location NAME.  One letter
; commands are left justified and padded on the right with zeros.  If the end
; of line is the next character, then this routine will return with NAME set
; to zero and no error.  If, however, there is a least one character out there
; and it is not a letter, then COMERR will be called...
NAMENW: CLA                     ; be sure the AC is zero
        DCA     NAME            ; and clear the resulting name
        .PUSHJ  @ZSPACMP        ; get the next character, whatever it is
        SNA                     ; is there anything there ??
         .POPJ                  ; no -- just give up now
        .PUSHJ  ALPHA           ; see if it is a letter
        SNL                     ; was it a letter ??
         JMP     @ZCOMERR       ; no -- this isn't legal
        TAD     ZMSPACE         ; yes -- convert it to SIXBIT
        BSW                     ; left justify it
        DCA     NAME            ; and store it in word

; Check for a second letter in the name...
```

```
        .PUSHJ  @ZGET           ; get the next character
        .PUSHJ  ALPHA           ; is this a letter ??
        SNL                     ; ???
         JMP     @ZBACKUP       ; no -- put it back and return
        TAD     ZMSPACE         ; yes -- convert it to SIXBIT too
        TAD     NAME            ; put both letters together
        DCA     NAME            ; ...
        .POPJ                   ; then that's all

;   This routine will return with the LINK bit set if the AC holds a letter,
; and with the LINK reset if it does not.  In either case the AC is not
; disturbed...
ALPHA:  STL                     ; be sure the link starts in a known state
        TAD     [-"A"]          ; compare it to the first letter
        SMA                     ; skip if it isn't a letter
         JMP     ALPHA1         ; it might be -- look further
        TAD     ["A"]           ; it's not a letter -- restore the AC
        .POPJ                   ; and quit (the link is zero now !!)

; Here if it might be a letter (the link is also zero now)...
ALPHA1: TAD     ["A"-"Z"-1]     ; now compare it to the other end
        SMA                     ; skip if it is a letter
         CML                    ; it isn't a letter -- set the link to a 0
        TAD     ["Z"+1]         ; restore the character and the link
        .POPJ                   ; then that's all
        .TITLE  Command Lookup and Dispatch


;   This routine will lookup a command or register name in a table and then
; dispatch to the corresponding routine.  The address of the table, should
; be passed in the AC.  The table is formatted as two word entries - the first
; word of a pair is the SIXBIT name of the command or register, and the second
; word is the address of the routine to call.  As soon as this routine finds a
; first word that matches the value currently in NAME, it will jump to the
; routine indicated by the second word.  The table ends with a zero word
; followed by the address of an error routine - the zero word always matches
; the current name and the error routine will be called.
;
;   NOTE: Command tables are always stored in field one, however the addresses
; of all the routines they reference are always in field zero!
;
;   NOTE: Auto index registers pre-decrement, so the address -1 of the table
; must be passed in the AC!
MATCH:  DCA     X1              ; save the pointer to the table
        CDF     1               ; command tables are stored in field 1

; Search for a name which matches...
MATCH1: TAD     @X1             ; pick up the name (from field 1)
        SNA                     ; is this the end of the table ??
         JMP     MATCH2         ; yes -- this always matches
        CIA                     ; make the word negative
        TAD     NAME            ; and compare it to the desired value
        SNA CLA                 ; ???
         JMP     MATCH2         ; a match !!
        ISZ     X1              ; no match -- skip over the address
        JMP     MATCH1          ; and keep looking

; Here when we find a match...
MATCH2: TAD     @X1             ; get the address of the routine
        DCA     MATCH3          ; put that in a safe place
        CDF     0               ; change back to the usual data field
        JMP     @MATCH3         ; then branch to the right routine

; Temporary storage for MATCH...
MATCH3: .BLOCK  1               ; address of the matching routine
        .TITLE  Scan Decimal Numbers


;   This routine will read a decimal number from the command line and return
; its value in location WORD.  The value is limited to 12 bits and overflows
; are not detected.  At least one decimal digit must be found on the command
; line or COMERR will be called, and the first non-digit character found will
```

```
; be returned in location SAVCHR.
DECNW:  CLA                     ; ignore the AC initially
        DCA     WORD            ; clear the total
        DCA     DIGITS          ; and the digit counter
        .PUSHJ  @ZSPACMP        ; ignore any leading blanks

; Check for a decimal digit...
DECNW1: TAD     [-"0"]          ; compare it to zero
        SPA                     ; ???
         JMP     DECNW2         ; it's not a digit -- quit
        TAD     [-9.]           ; then compare it to the other end
        SMA SZA CLA             ; ???
         JMP     DECNW2         ; still not a digit

; Accumulate another decimal digit...
        TAD     WORD            ; get the old total
        CLL RAL                 ; multiply it by two
        DCA     WORD            ; and save that
        TAD     WORD            ; ...
        CLL RAL                 ; then multiply it by 4 more
        CLL RAL                 ; (for a total of 8)
        TAD     WORD            ; because 8x + 2x = 10x
        TAD     SAVCHR          ; then add the new digit
        TAD     [-"0"]          ; and correct for ASCII characters
        DCA     WORD            ; remember that for next time

; Read the next digit and proceed...
        ISZ     DIGITS          ; remember one more digit processed
        .PUSHJ  @ZGET           ; get the next character
        JMP     DECNW1          ; then keep trying

; Here when we find something which isn't a digit...
DECNW2: CLA                     ; ...
        TAD     DIGITS          ; get the digit count
        SNA CLA                 ; it has to be at least one
         JMP     @ZCOMERR       ; that's an error if it isn't
        .POPJ                   ; and we're done
        .TITLE  Scan Octal Numbers


;   This routine will read an octal number from the command line and return
; its value in location WORD.  The value is usually limited to twelve bits,
; however any overflow bits will be left in location WORDH.  This is intended
; for use by the OCTNF routine to extract the field from a 15 bit address.
; At least one octal digit must be found on the command line or COMERR will be
; called, and the first non-digit character found will be returned in location
; SAVCHR.
OCTNW:  CLA                     ; remove any junk
        DCA     WORD            ; clear the partial total
        DCA     DIGITS          ; we haven't read any digits yet
        .PUSHJ  @ZSPACMP        ; ignore any leading spaces

; Check for an octal digit next...
OCTN1:  TAD     [-"0"]          ; compare it to a zero
        SPA                     ; ???
         JMP     OCTN2          ; this one isn't a digit
        TAD     [-7]            ; now compare to the high end of the range
        SMA SZA CLA             ; ???
         JMP     OCTN2          ; still not a digit

; Now accumulate another digit.
        TAD     WORD            ; get the previous total
        DCA     WORDH           ; and remember that for OCTNF
        TAD     WORD            ; then again
        RTL                     ; shift it left 3 bits
        RAL                     ; ...
        AND     [7770]          ; then insure that no junk has wrapped around
        TAD     SAVCHR          ; add the next digit
        TAD     [-"0"]          ; and correct for ASCII values
        DCA     WORD            ; remember the new total
        ISZ     DIGITS          ; also remember how many digits we read
        .PUSHJ  @ZGET           ; read the next character
```

```
        JMP     OCTN1           ; then go look for more

; Here when we find something which isn't a digit...
OCTN2:  CLA                     ; ...
        TAD     DIGITS          ; see how many digits we've read
        SNA CLA                 ; there must be at least one
         JMP     @ZCOMERR       ; nope -- this isn't legal
        .POPJ                   ; and return that in the AC

        .PAGE
        .TITLE  Scan 15 Bit Addresses


;   This routine will read a 15 bit address from the command.  The lower 12
; bits of the address are always left in location WORD, and the upper 3 bits
; will be in location ADRFLD, properly justified.  If the user types 5 or more
; digits in the octal address, the lower 12 bits becomes the address, and the
; next 3 most significant bits are the field.   If his octal number has 4 or
; fewer digits, the field will be the current data field instead.  For example,
; (assume that the current DF is 3):
;
;        1234 --> Location 1234, field 3
;       01234 --> Location 1234, field 0
;       41234 --> Location 1234, field 4
;     5641234 --> Location 1234, field 4
;
;   Like the OCTNW routine, this routine will return the low order 12 bits
; in location WORD. There is an alternate entry point at location OCTNI; this
; is identical to OCTNF, except that the instruction field, not the data field,
; provides the default field number...

; Here to read an address in the instruction field...
OCTNI:  CLA                     ; ...
        TAD     UFLAGS          ; use the instruction field as default
        AND     ZK70            ; ...
        JMP     OCTNF1          ; then proceed normally

; Here to read an address in the data field...
OCTNF:  CLA                     ; ...
        TAD     UFLAGS          ; use the data field as the default
        R3L                     ; ...
        AND     ZK70            ; ...
OCTNF1: DCA     ADRFLD          ; and save the default for later
        .PUSHJ  @ZOCTNW         ; read a normal octal number
        CLA                     ; we don't care about this part
        TAD     DIGITS          ; see how many digits there were
        TAD     [-5]            ; we need at least 5
        SPA CLA                 ; ???
         .POPJ                  ; there weren't that many -- use the default

; Extract the upper 3 bits of the address...
        TAD     WORDH           ; get the high order bits
        BSW                     ; then put the upper 3 bits in the right place
        AND     ZK70            ; trim it to just the important bits
        DCA     ADRFLD          ; then that is the new data field (temporarily)
        .POPJ                   ; that's all folks
        .TITLE  Ask For Confirmation


;   This routine will type a question mark and then wait for the operator to
; enter a "Y" (or "y") to confirm.  It's used by exceptionally dangerous
; commands, like FORMAT, and normally the caller will type a short string
; (e.g. "Format unit 0" before actually calling this function.  If the
; operator does confirm, it will return with the link set.  If the operator
; types anything other than "Y" or "y", it will return with the link clear.
; Note that it's also acceptable to just type Control-C to abort!
CONFRM: .PUSHJ  @[TQUEST]       ; type a question mark
CONF1:  .PUSHJ  @[INCHRS]       ; go get a character of input
        SNA                     ; did we get anything ?
         JMP     CONF1          ; nope - keep waiting
        DCA     SAVCHR          ; we got a real character - save it
        TAD     SAVCHR          ; and echo it back to the terminal
```

```
        .PUSHJ  @ZOUTCHR        ; ...
        .PUSHJ  @ZCRLF          ; followed by a CRLF

; See what his answer was...
        TAD     SAVCHR          ; get the answer once more
        TAD     [-"Y"]          ; is it a "Y"
        SNA                     ; ???
         JMP     CONF2          ; yes - return with the link set
        TAD     ["Y"-"y"]       ; or a "y"?
        SNA                     ; ???
         JMP     CONF2          ; yes - same thing
        CLA CLL                 ; nope - return FALSE
        .POPJ

; Here if he answered "Y" or "y"...
CONF2:  CLA STL                 ; return TRUE
        .POPJ                   ; ...
        .TITLE  Type Special Characters


;   This  routine will simulate a TAB on the terminal, which it does by typing
; spaces until the horizontal position reaches a multiple of 8 characters.
; Note that this routine will always type at least one space.  The AC is
; always cleared by this routine.
TTABC:  .PUSHJ  TSPACE          ; Always type at least one space
        TAD     HPOS            ; Get the current horizontal position
        AND     ZK7             ; Is it a multiple of 8 ??
        SZA CLA                 ; ???
         JMP     TTABC          ; No -- keep typing
        .POPJ                   ; Yes -- we can stop now

; This  routine will type a space on the terminal.
TSPACE: CLA                     ; Clear the AC
        TAD     [" "]           ; And load a space character
        JMP     @[THCHAR]       ; Then type it and return

; This routine will type a question mark on the terminal.
TQUEST: CLA                     ; ...
        TAD     ["?"]           ; ...
        JMP     @[THCHAR]       ; ...

; This routine will type a BELL character on the terminal.
TBELL:  CLA                     ; ...
        TAD     [CHBEL]         ; Get a bell character
        JMP     @[TFCHAR]       ; Then type it out

; Type a slash (used as an address separator) on the terminal.
TSLASH: CLA                     ; ...
        TAD     ["/"]           ; ...
        JMP     @[THCHAR]       ; ...

; Type a dot (used for decimal numbers and disk addresses) on the terminal.
TDOT:   CLA                     ; ...
        TAD     ["."]           ; ...
        JMP     @[THCHAR]       ; ...

; Convert the value in the AC to a decimal digit and type it...
TDIGIT: TAD     ["0"]           ; make it ASCII
        JMP     @[THCHAR]       ; type it and return
        .TITLE  Type Carriage Return, Line Feed and Backspace


;   This routine will type a carriage return, line feed pair on the terminal
; and it will correctly update HPOS to show that the cursor is now at the left
; margin.  In addition, it will keep count of the number of CRLFs output in
; location VPOS, and when the terminal's screen is full (as indicated by VPOS
; equals LENGTH) it will cause an automatic XOFF.  The AC is always cleared
; on return.
CRLF:   CLA                     ; be sure the AC is cleared
        TAD     [CHCRT]         ; get a return character
        .PUSHJ  @[TFCHAR]       ; and type that out
        DCA     HPOS            ; remember that the cursor is in column zero
```

```
; Now check the vertical position of the cursor...
        ISZ     VPOS            ; increment the current position
         NOP                    ; ...
        TAD     LENGTH          ; get the size of the screen
        SNA                     ; is it zero ??
         JMP     CRLF1          ; yes -- never stop
        CIA                     ; no -- make it negative
        TAD     VPOS            ; and compare it to the current location
        SPA CLA                 ; is the screen full ??
         JMP     CRLF1          ; no -- proceed
        DCA     VPOS            ; yes -- clear the vertical position
        .PUSHJ  @[TBELL]        ; type a bell character
        STA                     ; then load a -1 into the AC
        DCA     XOFF            ; and cause an automatic XOFF

; Type the line feed next...
CRLF1:  TAD     [CHLFD]         ; now get a line feed
        JMP     @[TFCHAR]       ; type that and return

;   This routine will type a BACKSPACE character on the terminal and update
; HPOS to show the new cursor position.  It will not allow you to backspace
; beyond the left margin of the temrinal. The AC is always cleared on return.
TBACKS: STA                     ; load the AC with -1
        TAD     HPOS            ; and decrement HPOS
        SPA                     ; are we going to pass the left margin ??
         JMP     BACKS1         ; yes -- don't type anything
        DCA     HPOS            ; no -- update HPOS
        TAD     [CHBSP]         ; then get a backspace character
        .PUSHJ  @[TFCHAR]       ; and type that out
BACKS1: CLA                     ; clear the AC
        .POPJ                   ; and that's all

        .PAGE
        .TITLE  Read Command Lines


;   This routine will read a single command line from the user and store the
; text of the line, one character per word and terminated by a null character,
; in the array at CMDBUF.  The size of CMDBUF, and therefore the maximum
; length of a command, is given by MAXCMD and is normally a page (128 words).
; An auto-index register, L, is set aside just for the purpose of indexing the
; command buffer and when it returns this routine will always leave L set up
; to point to the beginning of the command.
;
;   While it is reading the command, this routine will recognize these control
; characters:
;
;       Control-C --> Abort the command
;       Control-R --> Retype the current line, including corrections
;       Control-U --> Erase the current line and start over again
;       DELETE    --> Erase the last character (echos the last character typed)
;       BACKSPACE --> Erase the last character on a CRT
;       Return    --> Terminates the current command
;       Line Feed -->    "        "     "       "
;       ESCAPE    -->    "        "     "       "
;
;   When this routine is called, the AC should contain the prompting
; character.
;
INCHWL: DCA     PROMPT          ; remember the prompt character
        TAD     [CMDBUF-1]      ; point to the line buffer
        DCA     L               ; and initialize the pointer
        DCA     CMDLEN          ; say that this command is zero characters
        DCA     XOFF            ; clear the XOFF and
        DCA     CTRLO           ;  control-O flags...
        TAD     PROMPT          ; get the prompting address back again
        .PUSHJ  @ZOUTCHR        ; and type out the character

; Read and process the next character...
INCHW1: .PUSHJ  @[INCHRS]       ; try to read something from the console
        SNA                     ; did we get anything ??
```

```
         JMP     INCHW1          ; no -- wait for it
        DCA     SAVCHR          ; save this character for a while
        DCA     XOFF            ; then clear the XOFF,
        DCA     CTRLO           ;  control-O, and
        DCA     VPOS            ;  automatic XOFF flags
        TAD     SAVCHR          ; get the character back
        TAD     ZMSPACE         ; compare this to a space
        SPA                     ; ???
         JMP     INCHW3          ; this is a control character
        TAD     [" "-177]       ; is this a DELETE character ?
        SNA CLA                 ; ???
         JMP     INCHW8          ; yes -- go do that

; Here to process a normal character...
INCHW2: TAD     CMDLEN          ; get the length of this line
        TAD     [-MAXCMD]       ; and compare to the maximum
        SMA CLA                 ; are we already full ??
         JMP    INCH10          ; yes -- don't store this character
        TAD     SAVCHR          ; get the character back
        .PUSHJ  @ZOUTCHR        ; and echo it to the terminal
        TAD     SAVCHR          ; get the character again
        DCA     @L              ; store it in the line
        ISZ     CMDLEN          ; the command is one character longer now
        JMP     INCHW1          ; and go get the next one

; Here to handle a control-R command...
INCHW3: TAD     [" "-CHCTR]     ; is this really a control-R ??
        SZA                     ; ???
         JMP     INCHW4          ; no -- Proceed
        DCA     @L              ; yes -- close the command buffer
        TAD     [CHCTR]         ; get the control-R character back
        .PUSHJ  @ZOUTCHR        ; and echo that to the terminal
        .PUSHJ  @ZCRLF          ; start a new line
        TAD     PROMPT          ; get the prompt character first
        .PUSHJ  @ZOUTCHR        ; and always type that too
        TAD     [CMDBUF-1]      ; point to the current command
        .PUSHJ  @[TASCIZ]       ; and echo the entire line back
        .PUSHJ  @ZBACKUP        ; backup L over the null we put there
        JMP     INCHW1          ; finally continue typing

; Here to handle a Control-U character...
INCHW4: TAD     [CHCTR-CHCTU]   ; is this really a Control-U character ??
        SZA                     ; ???
         JMP     INCHW5          ; no -- keep trying
        TAD     [CHCTU]         ; yes -- get the character back again
        .PUSHJ  @ZOUTCHR        ; and echo that to the operator
        .PUSHJ  @ZCRLF          ; then start on a new line
        JMP     INCHWL+1        ; and go start all over again

; Here to handle a BACKSPACE character...
INCHW5: TAD     [CHCTU-CHBSP]   ; is that what this is ??
        SZA                     ; ???
         JMP     INCHW6          ; nope, not yet
        TAD     CMDLEN          ; yes -- get the length of this command
        SNA CLA                 ; is it a null line ??
         JMP     INCHW1          ; yes -- there's nothing to delete
        .PUSHJ  @[TBACKS]       ; yes, type a backspace
        .PUSHJ  @ZTSPACE        ; then type a space
        .PUSHJ  @[TBACKS]       ; and another backspace
        JMP     INCHW9          ; finally join with the DELETE code

; Here to check for line terminators...
INCHW6: TAD     [CHBSP-CHCRT]   ; is this a return ??
        SNA                     ; ???
         JMP     INCHW7          ; yes -- this line is done
        TAD     [CHCRT-CHLFD]   ; no -- Is it a line feed then ?
        SNA                     ; ???
         JMP     INCHW7          ; yes -- That's just as good
        TAD     [CHLFD-CHESC]   ; no -- How about an escape ?
        SZA CLA                 ; ???
         JMP     INCHW2          ; no -- just store this control character
```

```
; Here to finish a command...
        TAD     [CHESC]         ; get the ESCAPE code back
        .PUSHJ  @ZOUTCHR        ; and echo that to the terminal
INCHW7: .PUSHJ  @ZCRLF          ; then close the input line
        DCA     @L              ; end the command with a null byte
        TAD     [CMDBUF-1]      ; and then backup the pointer
        DCA     L               ;   to the start of the command
        .POPJ                   ; that's all there is to it

; Here to process a DELETE character...
INCHW8: TAD     CMDLEN          ; get the command length
        SNA CLA                 ; is this a null command ??
         JMP     INCHW1         ; yes -- there's nothing to delete
        .PUSHJ  @ZBACKUP        ; decrement the line pointer
        TAD     @L              ; get the last character stored
        .PUSHJ  @ZOUTCHR        ; and echo that for the DELETE

; Now delete the last character typed...
INCHW9: .PUSHJ  @ZBACKUP        ; decrement the line pointer
        STA                     ; then fix the command length too
        TAD     CMDLEN          ; ...
        DCA     CMDLEN          ; ...
        JMP     INCHW1          ; finally go get the next character

; Here if the command line is full -- echo a bell instead...
INCH10: .PUSHJ  @[TBELL]        ; go type a bell character
         JMP     INCHW1         ; then go wait for something to do

; Local storage for INCHWL...
CMDLEN: .BLOCK  1               ; the length of the current line
PROMPT: .BLOCK  1               ; the prompting character

        .PAGE
        .TITLE  Terminal Output Primitives


;   This routine will type the character in the AC on on the terminal.  If the
; character is a printing character, it will be typed normally.  If this
; character is happens to be a DELETE or NULL code (ASCII codes 00 and 7F),
; it will be ignored.  If the character is a TAB, it is simulated by calling
; the TTABC routine.  Finally, if it is any other control character, it is
; converted to the familiar ^x representation (unless it is an ESCAPE code,
; which, by tradition, is typed as $).  This routine cannot be used to type
; carriage returns, line feeds, bells, or other control characters that are
; to be output literally.  The AC is always cleared on return.
OUTCHR: SNA                     ; first see if this character is a null
         .POPJ                  ; just drop it if it is
        TAD     ZMSPACE         ; see if this is a control character
        SMA                     ; skip if it is a control code
         JMP     OUTCH3         ; just type a normal character

; Here to type a TAB character...
        TAD     [" "-CHTAB]     ; is this really a TAB character at all ??
        SZA                     ; ???
         JMP     OUTCH1         ; no -- check further
        JMP     @[TTABC]        ; yes -- type a TAB and return

; Here to print an ESCAPE character...
OUTCH1: TAD     [CHTAB-CHESC]   ; is this an ESCAPE character ??
        SZA                     ; ???
         JMP     OUTCH2         ; no -- go type the ^x form instead
        TAD     ["$"]           ; yes -- type a dollar sign for an ESCAPE
        JMP     THCHAR          ; then type it and return

; Here to print a control character...
OUTCH2: .PUSH                   ; save the character for a while
        CLA                     ; and get the flag character
        TAD     ["^"]           ; ...
        .PUSHJ  TFCHAR          ; type that first
        .POP                    ; then get the character back
        TAD     [CHESC+"@"]     ; convert it to a printing character
        JMP     THCHAR          ; type that and return
```

```
; Here to print a normal character...
OUTCH3: TAD     [" "]           ; restore the original character
                                ; and fall into THCHAR


;   This routine will type a printing character and, while doing this, it will
; keep track of the horizontal position of the cursor.  If it passes the line
; length of the terminal, a free carriage return is also typed.  The terminal's
; horizontal position (HPOS) is also used for the tab simulation.  The
; character to be typed should be in the AC, the AC will be cleared on return.
THCHAR: .PUSH                   ; save the character for a while
        ISZ     HPOS            ; and incrment the horizontal position
        CLA                     ; get the maximum width allowed
        TAD     WIDTH           ; ...
        SNA                     ; is it zero ??
         JMP     THCHA1         ; yes -- no automatic carriage returns, then
        CMA CIA                 ; make it negative
        TAD     HPOS            ; and compare to the terminal cursor position
        SPA CLA                 ; have we reached the end of the line ??
         JMP     THCHA1         ; no -- proceed normally
        .PUSHJ  @ZCRLF          ; yes -- force a carriage return first
THCHA1: .POP                    ; then get the character back
                                ; and fall into TFCHAR


;   This routine will type a single character from the AC on the terminal.
; Before it types the character, this routine will check the state of the
; CNTRLO and XOFF flags. If a Control-O has been typed, the character is
; discarded and not typed on the terminal.  If an XOFF has been typed, the
; output will be suspended until the user types an XON character (or a
; Control-O or Control-C).
TFCHAR: .PUSH                   ; save the character for a while
TFCHA1: .PUSHJ  INCHRS          ; check the operator for input
        CLA                     ; we don't care if anything was typed
        TAD     CTRLO           ; get the Control-O flag byte
        SZA CLA                 ; is it zero ??
         JMP     TFCHA2         ; no -- just throw this character away
        TAD     XOFF            ; now test the XOFF flag
        SZA CLA                 ; is the output suspended ??
         JMP     TFCHA1         ; wait for something to happen if we are XOFFed

; Here when it is OK to type the character...
        .POP                    ; get the character back
        JMP     CONOUT          ; and send it to the UART

; Here to return without typing anything...
TFCHA2: .POP                    ; clean up the stack
        CLA                     ; but always return zero
        .POPJ                   ; and just quit

;   This routine will output a character from the AC to the terminal, with no
; no special processing of any kind.  It simply waits for the console flag to
; set and then send the character.  However, If the flag does not set in a
; reasonable amount of time then this routine will force the character out
; anyway.  This prevents the monitor from hanging if the terminal flag is
; cleared by the user's program.
;
;   The timeout loop requires 26 minor cycles which, with a 4.9152Mhz clock,
; takes 10.5 microseconds.  If we simply clear the timeout counter when we
; start we'll get a timeout after 4096 counts, or about 43 milliseconds.
; If we assume that 300 baud is the slowest console we'll ever use, then
; that's just about right (at 300 baud a character takes about 33 milliseconds
; to transmit!).
;
CONOUT: DCA     CONCHR          ; remember the character to send
        DCA     IRMA            ; and clear the timeout timer

; See if the flag is set and send the character if so...
CONOU1: TSF                     ; [9] is the flag set ???
         JMP     CONOU3         ; [4] no -- go check the timeout
CONOU2: TAD     CONCHR          ; yes -- get the character
```

```
        TLS                     ; and send it to the console
        CLA                     ; a _real_ TLS doesn't clear the AC!!
        .POPJ                   ; ...

; Here if the flag is not yet set...
CONOU3: ISZ     IRMA            ; [9] have we waited long enough  ???
         JMP     CONOU1         ; [4] no -- wait a little longer
        JMP     CONOU2          ; yes -- force the character out anyway

; Temporary storage for the CONOUT routine...
CONCHR: .BLOCK  1               ; a place to save the console character
        .TITLE  Terminal Input Primitives


;   This routine is called to check for operator input.  It will test to see
; if the operator has typed a character. If he has not, this routine returns
; with the AC cleared and nothing else happens. If he has, this routine checks
; to see if the character is one of Control-C, Control-O, Control-S or
; Control-Q because these characters have special meaning and are acted upon
; immediately. If the input character is anything else, the character is
; returned in the AC.
INCHRS: .PUSHJ  CONIN           ; try to read a character from the terminal
        AND     ZK177           ; ignore the parity bit here
        SNA                     ; is this a null character ??
         .POPJ                  ; yes -- just ignore it

; Here process a control-C character -- restart the monitor...
        TAD     [-CHCTC]        ; is this really a control-C ??
        SZA                     ; ???
         JMP     INCHR1         ; no -- proceed
        DCA     CTRLO           ; yes -- clear the control-O
        DCA     XOFF            ;  and XOFF flags
        TAD     [CHCTC]         ; get another control-C character
        .PUSHJ  @ZOUTCHR        ; echo it to the terminal
        JMP     @ZRESTA         ; and go restart the monitor

; Here to check for a control-O character...
INCHR1: TAD     [CHCTC-CHCTO]   ; compare to a control-O character
        SZA                     ; is this it ??
         JMP     INCHR2         ; no -- keep checking
        DCA     XOFF            ; control-O always clears the XOFF flag
        TAD     [CHCTO]         ; get another control-O character
        .PUSHJ  @ZOUTCHR        ; and echo that
        .PUSHJ  @ZCRLF          ; then close the line
        TAD     CTRLO           ; get the current state of the control-O flag
        CMA                     ; and complement it
        DCA     CTRLO           ; that's the new value
        .POPJ                   ; return with the AC cleared

; Here to check for a control-S character...
INCHR2: TAD     [CHCTO-CHXOF]   ; is this a control-S ??
        SZA                     ; ???
         JMP     INCHR3         ; nope, try again
        STA                     ; yes -- get a -1 into the AC
        DCA     XOFF            ; and set the XOFF flag
        .POPJ                   ; return with the AC cleared

; Here to check for a control-Q character...
INCHR3: TAD     [CHXOF-CHXON]   ; is this a control-Q ??
        SZA                     ; ???
         JMP     INCHR4         ; no -- just give up
        DCA     XOFF            ; yes -- clear the XOFF flag
        DCA     VPOS            ; also clear the automatic XOFF counter
        .POPJ                   ; return with the AC cleared

; Here if the character is nothing special...
INCHR4: TAD     [CHXON]         ; restore the state of the AC
        .POPJ                   ; and return the character in the AC


;   This routine will read a  single character from the console UART.  If no
; character is currently ready, it will return a null (zero) byte in the AC,
```

```
; but otherwise the character read is left in the AC...
CONIN:  CLA                     ; be sure the AC is cleared
        KSF                     ; is a character ready ??
         .POPJ                  ; no -- just return zero
        KRB                     ; yes -- read it into the AC
        .POPJ                   ; then return that in the AC

        .PAGE
        .TITLE  Control Panel Entry Points

        .ORG    7600

;   There's a little bit of chicanery that goes on here (when have you seen
; a PDP-8 program without that???).  After a power on clear or a hard reset,
; the HM6120 starts executing at location 7777 of panel memory which, in
; the case of the SBC6120, is part of the EPROM.  The EPROM code at this
; location always jumps to the system initialization routine without even
; trying to figure out why we entered panel mode.
;
;   The system initialization code copies all of the EPROM contents to panel
; RAM and then disables the EPROM forever.  After that it actually changes
; the vector at location 7777 to point to the CPSAVE routine, which is the
; normal panel entry point for traps, halts, etc.
        .VECTOR CPBOOT          ; set the 6120 start up vector at 7777
CPBOOT: CXF     1               ; the startup code lives in field 1
        JMP     @[SYSINI]       ; and away we go!


;   Excluding a hardware reset, the 6120 will enter control panel mode for any
; of three other reasons:
;
;  * any of the PR0..PR3 instructions were executed in main memory
;  * the CPU was halted, either by a HLT instruction or by the RUN/HLT input
;  * a panel interrupt was requested by the CPREQ pin
;
;   In all the these cases, the 6120 was presumably executing some important
; program in main memory before it was interrupted, and we need to save the
; state of that program before doing anything else.  When the 6120 enters
; panel mode it saves the last main memory PC in panel memory location 0 and
; then starts executing instructions in panel memory at 7777. The remainder of
; the main memory context (e.g. AC, MQ, flags, etc) we have to save manually.
CPSAVE: DCA     UAC             ; save the AC
        GCF                     ; and the flags (including LINK, IF and DF)
        DCA     UFLAGS          ; ...
        MQA                     ; the MQ
        DCA     UMQ             ; ...
        RSP1                    ; 6120 stack pointer #1
        DCA     USP1            ; ...
        RSP2                    ;  "  "   "  #2
        DCA     USP2            ; ..

;   Now set up enough context so that this monitor can run.  The CONT routine
; has saved in location STKSAV our last stack pointer before the main memory
; program was started and, if we're single stepping the main memory program,
; we're going to need that so that we can continue what we were doing.  In
; the case of other traps the RESTA routine gets called which will reset the
; stack pointer.
        TAD     STKSAV          ; get the monitor's last known stack pointer
        LSP1                    ; and restore that
        CXF     0               ; set both DF and IF to field zero
        SPD                     ; make indirect cycles access panel memory
        DCA     VPOS            ; reset the automatic XOFF line counter
        POST+1                  ; show post code #1

;  Finally, we can determine the exact reason for entry into panel mode by
; reading the panel status flags with the PRS instruction, and that will tell
; us where to go next.  Be careful, though, because executing PRS clears the
; flags so we only get to do it once!  This code kind of assumes that only one
; of these flags can be set at any time - I believe that's true for the 6120.
        PRS                     ; get the reason for panel entry
        RAL                     ; check the BTSTRP flag first
        SZL                     ;  ... this is set by an external CPREQ
```

```
         JMP     BTSTRP          ; yes
        RAL                      ; the next flag is PNLTRP
        SZL                      ;  ... which is set by the PRn instructions
         JMP     PNLTRP          ; ...
        RTL                      ; next is PWRON (it skips a bit!)
        SZL                      ;  ... which is only set by a hard reset
         JMP     PWRON           ; ...
        RAL                      ; and lastly is the HLTFLG
        SZL                      ;  ... which is set any time the CPU halts
         JMP     HALTED          ; ...

;   If we get here, none of the known panel status bits are set.  I don't
; know what this means, but it can't be good!  We also jump here if the
; PWRON status bit is set.  Since this bit can only be set by a hardware
; reset, and since in the SBC6120 this automatically maps EPROM instead
; of RAM, we should never see this happen.
PWRON:  JMS     TRAP            ; print a generic
          TRPMSG                ;   "% UNKNOWN TRAP AT ..." message

;   The BTSTRP flag indicates a transition of the CPREQ line.  In the
; SBC6120 this is conected to the console UART framing error output, so
; entering a BREAK on the terminal causes a console trap.  In other hardware
; this might also be used to trap various IOT instructions and emulate them,
; but not in the SBC6120...
BTSTRP: JMS     TRAP            ; print
         BRKMSG                 ;  "% BREAK AT ..." and restart

;   The PNLTRP flag indicates that one of the PR0 thru PR3 instructions has
; been executed, but unfortunately the only way to find out which is to
; use the last main memory PC to fetch the instruction from memory.  Remember
; that the 6120 will have already incremented the PC by the time we get here,
; so it's actually one _more_ than the location we want.  Currently the PR3
; instruction is used as a breakpoint trap and PR0 is a generic ROM "monitor
; call".  The other two, PR1 and PR2, are unused.
PNLTRP: STA                     ; decrement the PC
        TAD     UPC             ; so it points at the actual instruction
        DCA     UPC             ; that caused the trap
        TAD     UFLAGS          ; get the IF at the time of the trap
        AND     ZK70            ; ...
        TAD     PNLCDF          ; make a CDF instruction out of that
        DCA     .+1             ; and execute it
         NOP                    ; ... gets overwritten with a CDF ...
        CPD                     ; address main memory with indirect cycles
        TAD     @UPC            ; get the opcode that caused the trap
        DCA     UIR             ; and save it for later
        SPD                     ; back to panel memory
PNLCDF: CDF     0               ; always field zero

; See which instruction it was...
        TAD     UIR             ; get the opcode
        TAD     [-BPT]          ; is it PR3 ??
        SNA                     ; ???
         JMP     BPTTRP         ; yes - handle a break point trap
        TAD     [BPT-PR0]       ; no - is it PR0?
        CXF     1               ; (the ROM call handler lives in field 1)
        SNA                     ; ???
         JMP     @[MCALL]       ; yes - handle a monitor call
        CXF     0               ; ...
        JMS     TRAP            ; for any others just print a generic
         PRNMSG                 ;   "% PANEL TRAP AT ..." message

; Here for a breakpoint trap...
BPTTRP: JMS     TRAP            ; print
         BPTMSG                 ;  "% BREAKPOINT AT ..." and proceed

; Here (from field 1) for an illegal PR0 call...
ILLPR0: JMS     TRAP            ; say
         PR0MSG                 ; "? ILLEGAL PR0 FUNCTION AT ..."

;   We get here when the 6120 halts, but unfortunately there are no less than
; three different reasons why it night have done this.  The first is that the
; main memory program has executed a HLT (7402, or any microcoded combination
```

```
; there of) instruction.  Or, it could be that the 6120 was halted externally
; by a transition on the HLTREQ input pin, however the SBC6120 has no hardware
; to do this.  Lastly, it could be that the HALT flag was already set when
; we restarted the main memory program - in this case the 6120 will execute
; one instruction and trap back here.
;
;   We use this situation intentionally to single step main memory programs,
; and we can tell when this happens by checking the SIMFLG flag in memory.
; This flag is normally cleared, but will be set by the SINGLE routine when we
; want to single step.  In that case the monitor's stack is valid (it was
; saved to STKSAV by the CONT routine before switching context) and all we
; have to do is execute a .POPJ to return to the routine that originally
; called SINGLE.  Keep your fingers crossed.
HALTED: CLA                         ; ...
        TAD     SIMFLG              ; did we execute a single instruction?
        SZA CLA                     ; ???
         .POPJ                      ; yes - return from the SINGLE routine now!
        JMS     TRAP                ; otherwise just say
         HLTMSG                     ;  "% HALTED AT ..." and restart

;   This routine does most of the generic work of handling traps to panel
; memory.  It prints a message, which is passed inline via a JMS instruction,
; prints the PC, removes any breakpoints from the program and then restarts
; the monitor...
TRAP:   0                           ; call here with a JMS instruction
        .PUSHJ  @ZCRLF              ; be sure we start on a new line
        TAD     @TRAP               ; get the address of the message
        .PUSHJ  @[OUTSTR]           ; and print that
        TAD     UFLAGS              ; then get the field of the trap
        AND     ZK70                ; ...
        .PUSHJ  @[TFIELD]           ; and type that
        TAD     UPC                 ; then the PC too
        .PUSHJ  @ZTOCT4C            ; type that and a CRLF
        .PUSHJ  @[REGLSC]           ; type the registers on the next line
        .PUSHJ  @[BPTRMV]           ; remove any breakpoints
        JMP     @ZRESTA             ; and restart the monitor

        .FIELD  1
        .TITLE  Field 1 Variables


;   This page defines all the page zero variables used by the code in field
; one.  The system initialization code, part 1, at SYSINI: also lives in page
; zero of field one, and then is overwritten by these variables after init-
; ialization is completed.  As a consequence, none of these variables can
; have initial values the way their field zero counter parts do!

        .ORG    0000

; Auto index registers...
        .ORG    0010
RAMPTR: .BLOCK  1       ; address, within the RAM disk, for I/O
BUFPTR: .BLOCK  1       ; address of the caller's buffer for I/O
XX1:    .BLOCK  1       ; generic auto index register for field 1
XX2:    .BLOCK  1       ;   "       "    "      "      "    "
        .ORG    0020

; RAM Disk I/O routine storage...
RDUNIT: .BLOCK  1       ; currently selected RAM disk unit for I/O
RDPAGE: .BLOCK  1       ;   "    "    "    "    "    "  page number
RAMBUF: .BLOCK  3       ; a three byte "mini buffer" for 3 <-> 2 packing
RAMDAR: .BLOCK  1       ; RAM disk address register (written to LDAR)
BATTOK: .BLOCK  1       ; -1 if RAM backup battery is good
RDSIZE: .BLOCK  4       ; size of each RAM disk unit, in KB, or 0 if none
RAMSIZ: .BLOCK  1       ; total size of all RAM disks, in KB
SIZPTR: .BLOCK  1       ; pointer to the RDSIZE array
RAMUSZ: .BLOCK  1       ; - size of selected RAM disk chip


; IDE Disk I/O routine storage...
DKPART: .BLOCK  1       ; 12 bit disk partition number
DKRBN:  .BLOCK  1       ; 12 bit sector relative block number
```

```
DKSIZE: .BLOCK  1       ; size of attached drive, in MB, or 0 if no drive
DKUNIT: .BLOCK  1       ; logical unit (partition) number for OS/8
PARMAP: .BLOCk  10      ; unit number to partition map for eight OS/8 units

; ROM call arguments
MUUO:   .BLOCK  1       ; ROM MCALL function code
ARGPTR: .BLOCK  1       ; pointer to MCALL (PR0) argument list
XFRCNT: .BLOCK  1       ; word count for I/O
BUFPNL: .BLOCK  1       ; -1 if the user buffer is in panel memory
BUFSIZ: .BLOCK  1       ; actual buffer size for RDIBUF/WRIBUF
RWCNT:  .BLOCK  1       ; number of pages to be transferred

        .PAGE
        .TITLE  ROM Calls (PR0 Instructions)

        .ORG    0200

;   The PDP2HEX program (which converts BIN files into ROM images in Intel
; HEX format) stores a checksum of ROM field 1 in location 10200.  This is
; used by the POST and the VE (version) command.
ROMCK1: .BLOCK  1

;   This routine is called by CPSAVE when it detects a panel entry caused by
; a PR0 instruction.  Main memory programs can use this instruction to
; communicate with the ROM firmware and, in particular, the OS/8 device driver
; for the RAM disk uses PR0 to transfer data.  At this point all of the main
; memory program's registers have been saved and our original monitor stack
; has been restored.  The data and instruction field are both one and the
; 6120 panel data flag is set (so indirect references go to panel memory).
; That's about all we can depend on.
;
;   There are a couple of subtle points to watch out for here.  One simple
; one is that, to save time, break points are not removed from the caller's
; program while we interpret a PR0.  That means we have to be sure and return
; to main memory by jumping to CONT1, not CONT, since the latter will attempt
; to reinstall breakpoints _again_ and forever loose the original contents
; of those locations.
;
;   The other thing to remember is that CONT and CPSAVE conspire to preserve
; the monitor's stack, so that it can return to the correct place while single
; stepping.  That means we want to be sure and JMP to CONT1, not .PUSHJ to it,
; because otherwise it'll just return back to us the next time we enter panel
; mode!
;
;   The convention is that the first word after PR0 is a function code to
; select the firmware routine.   This routine also preserves the contents of
; the AC both ways - that is, whatever was in the user's AC when the PR0 was
; executed will be in the AC when our monitor call function is invoked, and
; whatever our monitor call function returns in the AC will be placed in the
; user's AC when control returns from the PR0.  Anything more than that is up
; to the specific function invoked.

;   Get the first agument (the function code) and use it to determine the
; address of a ROM routine to handle it...
MCALL:  .PUSHJ  GETARG          ; CPSAVE leaves the PC pointing at the PR0
                                ;  so do a dummy GETARG to skip it
        .PUSHJ  GETARG          ; then get a real PR0 argument from main memory
        DCA     MUUO            ; this is always the function code
        TAD     MUUO            ; see if it's in range
        CLL                     ; be sure the link is in a known state
        TAD     [-MAXFUN-1]     ; check against the maximum function
        SZL CLA                 ; if it's legal, skip
         JMP     MCALL2         ; nope - go print an error message...
        TAD      MUUO           ; it's legal - use it
        TAD     [FUNTBL]        ; index into the function dispatch table
        DCA     MUUO            ; ...
        TAD     @MUUO           ; get the address of the function routine
        DCA     MUUO            ; finally - that's what we wanted to know!

;   Invoke the monitor routine, preserving the AC in both directions.  In
; addition, the LINK bit is commonly used as an error flag (i.e. the LINK
; set on return indicates an error), so it is preserved on return only.
```

```
        CDF     0               ; the user's context lives in field 0
        TAD     @[UAC]          ; get the user's AC
        CDF     1               ; all ROM call routines live in field 1
        .PUSHJ  @MUUO           ; call routine to execute the ROM call
MCALL1: CDF     0               ; address the user's context again
        DCA     @[UAC]          ; return whatever's in the AC
        RAR                     ; put the link bit in AC0
        DCA     MUUO            ; save it for a minute
        NL3777                  ; then mask off the LINK bit
        AND     @[UFLAGS]       ; in the user's flags
        TAD     MUUO            ; and put ours in there instead
        DCA     @[UFLAGS]       ; ...
        CXF     0               ; CONT1 lives in field 1
        JMP     @[CONT1]        ; and then return to main memory

; Here when an illegal PR0 function is invoked.
MCALL2: CXF     0               ; say
        JMP     @[ILLPR0]       ;  "?Illegal PR0 function at ..."
        .TITLE  Fetch PR0 Arguments


;   This routine fetches an argument for PR0 from the main memory program.
; Since arguments are always stored in line after the PR0, the next argument
; is in the instruction field and pointed to by the last main memory PC.
; After the argument is fetched the main memory PC is always incremented so
; that we'll skip over the argument when we return - you have to be careful
; about this, since it means this routine can only be called ONCE to fetch
; any given argument!  The PR0 argument is returned in the AC.
GETARG: CLA                     ; just in case
        CDF     0               ; BEWARE - UFLAGS and UPC are both in field 0!
        TAD     @[UPC]          ; get the user's PC from field 0
        DCA     ARGPTR          ; save it in field 1 for a moment
        ISZ     @[UPC]          ; and increment it to skip over the argument
         NOP                    ;  this really shouldn't ever happen!
        TAD     @[UFLAGS]       ; get the last known user (main memory) flags
        AND     [70]            ; then get the IF at the time of the trap
        TAD     [CDF 0]         ; make a CDF instruction
        DCA     .+1             ; change to the correct field
         NOP                    ; ... gets overwritten with a CDF ...
        CPD                     ; always fetch from main memory
        TAD     @ARGPTR         ; get the next word from user program space
        SPD                     ; back to panel memory
        CDF     1               ; and back to our field
        .POPJ                   ; return the PR0 argument in the AC
        .TITLE  ROM Call Table


;   This is what you've really been waiting for - the table of ROM firmware
; function codes and routine addresses.
FUNTBL: GETVER                  ;  0 - get ROM version
        RAMDRW                  ;  1 - read/write RAM disk
        GETRDS                  ;  2 - return RAM disk size
        GETBAT                  ;  3 - return RAM disk battery status
        DISKRW                  ;  4 - read/write IDE disk
        GETDKS                  ;  5 - return IDE disk size
        SETPMP                  ;  6 - set disk partition mapping
        GETPMP                  ;  7 - get disk partition mapping
        MEMMOV                  ; 10 - copy memory
MAXFUN=.-FUNTBL-1


; PR0 function zero returns the current firmware version in the AC...
GETVER: CLA CLL                 ; ...
        TAD     [VERSION]       ; get our version number
        .POPJ                   ; MCALL will store it in the caller's AC

        .TITLE  Return from Routines in Field 1


;   This routine is the other half of the code at PUSHJ1:, and it allows
; routines in field one which were called from field zero to return to
; field zero.  It only needs to do two instructions, but those instructions
```

```
; have to be somewhere in field one!
POPJ1:  CXF     0               ; return to field zero
        .POPJ                   ; the address is already on the stack
        .TITLE  RAM Disk support


;   The SBC6120 contains a DS1221 SRAM controller with Li battery backup and
; sockets for up to four byte wide SRAM chips.  Each socket can contain either
; a HM628512 512Kx8 SRAM or a HM628128 128Kx8 SRAM or, of course, nothing.
; Additionally, the last socket has two jumpers which permit a 512K byte
; CMOS EPROM to be used if desired.  The maximum capacity of the RAM disk
; array is thus 2Mb - a pretty respectable sized disk (almost as big as a
; RK05J!) for OS/8.
;
;   The SBC6120 maps these RAM chips into panel memory via the memory decode
; GAL and, when memory map 3 (MM3) is enabled, all indirect references to panel
; memory will access the RAM disk array.  Since the RAM disk is only a byte
; wide, write operations discard the upper four bits of a twelve bit word, and
; when reading these bits are undefined and should be masked off by the
; software.
;
;   Addressing the RAM disk is a little tricky, since a 2Mb memory requires
; a total of 21 address bits - quite a bit more than a PDP-8 can manage.
; RAM disk address bits 0..11 (the low order bits, contrary to the PDP-8
; convention) are supplied by the HM6120 MA11-0.  The remaining 7 bits needed
; by each 512K SRAM come from a special register, the Disk Address Register,
; which can be loaded via the LDAR IOT.  The final two bits needed by the
; DS1221 to select one of the four SRAM chips come from DF0 and DF1 (DF2 is
; not used at the moment).
;
;   Put more simply, the DF selects the SRAM chip used, the DAR selects the
; 4K byte "bank" within the chip, and the normal memory address selects the
; individual byte within the bank.
;
;   For the purposes of writing an OS/8 device handler, each 4K RAM disk bank
; contains 21 pages of 128 twelve bit words, packed using the standard OS/8
; "three for two" scheme.  A 512K SRAM chip can hold 128 of these banks,
; corresponding to DAR addresses 0..127, for a total capacity of 2688 PDP-8
; pages or 1344 OS/8 blocks.  A 128K SRAM would contain only 32 banks, for a
; total of 672 PDP-8 pages or 336 OS/8 blocks.
;
;   Sixty-four bytes are wasted in each bank by this packing scheme, which
; works out to about 21 OS/8 blocks lost in a 512K SRAM.  More clever software
; could reclaim these, but it would require that the three-for-two packing
; algorithm split PDP-8 pages across RAM disk banks.
;
;   The SRAMs are optional, and this SBC6120 may have all, only some, or even
; none installed.  Since each SRAM chip is treated as a separate OS/8 unit,
; this makes it easy to handle the situation where some chips are not missing -
; these units are simply "off line".

; RAM disk "geometry" constants...
RAM512=2688.    ; size of a 512K RAM disk, in pages
RAM128=672.     ;  "   "  " 128K  "    "    "   "
BANKSZ=21.      ; pages per bank of RAM disk memory

; Special IOTs for the RAM disk hardware...
LDAR=6410       ; Load RAM disk address register
        .TITLE  RAM Disk Read/Write ROM Call


; The calling sequence for the PR0 RAM disk R/W function is:
;
;       PR0
;        0001           / panel function code for RAMDISK I/O
;        <arg1>         / R/W bit, page count, buffer field and unit
;        <arg2>         / buffer address
;        <arg3>         / starting page number (not block number!)
;       <return>        / AC == 0 if success; AC != 0 if error
;
; The error codes currently returned by RAMDRW are:
;
```

```
;       0001 - unit > 3 or SRAM chip not installed
;       0002 - page number > 2688
;
; If this looks a lot like an OS/8 handler call, that's no accident!
RAMDRW: .PUSHJ  @[SETBUF]       ; set up MUUO, BUFPTR, BUFCDF and RWCNT
        .PUSHJ  GETARG          ; and lastly get the disk page number
        DCA     RDPAGE          ; ...

; Select (after first ensuring that it exists!) the correct unit...
        TAD     MUUO            ; next get the unit number
        AND     [7]             ; ...
        DCA     RDUNIT          ; ...
        .PUSHJ  @[RAMSEL]       ; setup RAMCDF to select the correct "unit"
        SZL CLA                 ; was the unit number illegal ?
         JMP     @[RAMER1]      ; yes - give the error return

; This loop reads or writes pages 'till we've done all we're supposed to...
RDRW1:  .PUSHJ  @[SETDAR]       ; calculate the RAM disk address and bank
        SZL CLA                 ; was the page number valid?
         JMP     @[RAMER2]      ; nope - give the bad page error return
        TAD     MUUO            ; get the function code again
        SMA CLA                 ; should we read (0) or write (1) ?
         JMP     RDRW2          ;  ... read
        .PUSHJ  @[PACK]         ; transfer a page from memory to RAM disk
        JMP     RDRW3           ; and continue
RDRW2:  .PUSHJ  @[UNPACK]       ; transfer a page from RAM disk to memory
RDRW3:  ISZ     RDPAGE          ; if we need more, continue on the next page
        ISZ     RWCNT           ; have we done enough pages?
         JMP     RDRW1          ; nope - keep going
        CLA                     ; all done with the RAMDRW call
        .POPJ                   ; return status code zero (no error)
        .TITLE  RAM Disk Primary Bootstrap


;   This routine will read page zero from RAM disk unit zero into page
; zero of field zero of main memory.  The next step in the usual boot
; sequence would be to start the secondary bootstrap, but that's up to
; the caller...
RDBOOT: STA                     ; point the buffer to page 0
        DCA     BUFPTR          ; ...
        TAD     [CDF 0]         ; of field zero
        DCA     @[BUFCDF+1]     ; ...
        DCA     BUFPNL          ; of main memory
        DCA     RDUNIT          ; read RAM disk unit zero
        DCA     RDPAGE          ; page zero
        JMP     @[RAMDRD]       ; ...

        .PAGE
        .TITLE  Read and Write RAM Disk Pages


;   This routine will read a single page from RAM disk to a buffer in memory.
; The caller must set up RDUNIT and RDPAGE with the desired RAM disk unit
; and page, and BUFPTR, BUFCDF and BUFPNL with the address of a 128 word
; buffer in 6120 memory.  If any errors are encountered, this routine will
; return with the LINK set and an error status in the AC.
RAMDRD: .PUSHJ  @[RAMSEL]       ; select the unit in RDUNIT
        SZL                     ; was it invalid??
         JMP     RAMER1         ; yes - return error code 1
        .PUSHJ  @[SETDAR]       ; calculate the necessary disk address
        SZL                     ; is the page number invalid?
         JMP     RAMER2         ; yes - return error code 2
        JMP     @[UNPACK]       ; unpack RAM disk data to the buffer and return


;   This routine will write a single page from 6120 memory to RAM disk.  Except
; for the direction of data flow, it's identical to RAMDRD, including all the
; parameters and error returns.
RAMDWR: .PUSHJ  @[RAMSEL]       ; select the unit
        SZL                     ; was it invalid??
         JMP     RAMER1         ; yes - return error code 1
        .PUSHJ  @[SETDAR]       ; calculate the disk address
```

```
        SZL                     ; invalid page number??
         JMP     RAMER2         ; yes - return error code 2
        JMP     @[PACK]         ; pack buffer data into the RAM disk and return


; Here if the unit number is invalid...
RAMER1: CLA CLL CML IAC         ; return LINK = 1 and AC = 1
        .POPJ                   ; ...

; Here if the page number is invalid...
RAMER2: NL0002                  ; return AC = 2
        STL                     ; and LINK = 1
        .POPJ                   ; ...
        .TITLE  Unpack RAM Disk Pages


;   This routine will read one page (aka a sector) of 128 PDP-8 words from
; the RAM disk to a buffer anywhere in main memory or panel memory.  The
; address of the disk page read is selected by the RAMCDF and RAMPTR locations
; and the DAR register, which should be set up by prior calls to the RAMUNI
; and SETDAR routines.  The address of the buffer written is selected by the
; BUFPTR, BUFCDF and BUFPNL locations, which must be set up by the caller
; before invoking this routine.  Exactly 128 words are always transferred,
; without fail!
UNPACK: CLA                     ; ...
        TAD     [-64.]          ; one page is 128 words, or 64 word pairs
        DCA     XFRCNT          ; ...

;   Fetch the next three bytes (two words) from the SRAM chip.  Note that
; the SRAMs are only eight bits wide, so we'll read indeterminate garbage for
; the upper four bits of each word.  Those have to be masked off on the first
; two bytes, but for the third one it doesn't matter - it gets masked to two
; four bit pieces later anyway...
UNPAC1: JMS     @[RAMCDF]       ; change the DF to the RAM disk unit
        MM3                     ; and enable the RAM disk chips
        TAD     @RAMPTR         ; fetch three bytes from RAM disk
        AND     [377]           ; eight bits only, please
        DCA     RAMBUF          ; ...
        TAD     @RAMPTR         ; ...
        AND     [377]           ; ...
        DCA     RAMBUF+1        ; ...
        TAD     @RAMPTR         ; ...
        DCA     RAMBUF+2        ; ...
        MM2                     ; restore the default memory map
        CDF     1               ; and field

; Pack the three bytes into two words and store them in main/panel memory...
        JMS     @[BUFCDF]       ; change DF to the buffer field
        TAD     RAMBUF+2        ; the upper 4 bits of the first word are here
        BSW                     ; shift them left six
        CLL RTL                 ; ... then eight bits
        AND     [7400]          ; and isolate just those four bits
        TAD     RAMBUF          ; assemble the first word
        DCA     @BUFPTR         ; and store it in main memory
        TAD     RAMBUF+2        ; now do the upper 4 bits of the second word
        CLL RTL                 ; shift them left two
        CLL RTL                 ; ... then four bits
        AND     [7400]          ; and isolate just those four bits
        TAD     RAMBUF+1        ; reassemble the second word
        DCA     @BUFPTR         ; store that in main memory too
        SPD                     ; return to panel memory
        CDF     1               ; and our own memory field
        ISZ     XFRCNT          ; have we done a full page?
         JMP     UNPAC1         ; nope - keep copying
        CLL                     ; be sure the LINK is cleared for success
        .POPJ                   ; yes - we're outta here!
        .TITLE  Pack RAM Disk Pages


;   This routine will write one page of 128 PDP-8 words from a buffer anywhere
; in either panel or main memory to RAM disk.  It's the exact complement of
; UNPACK, and expects exactly the same things to be set up.
```

```
PACK:   CLA                     ; don't assume anything!
        TAD     [-64.]          ; do 64 word pairs, or 128 words
        DCA     XFRCNT          ; ...

; Grab the next two twelve bit words from the buffer...
PACK1:  JMS     @[BUFCDF]       ; change DF to the buffer's field
        TAD     @BUFPTR         ; get a word from the buffer
        DCA     RAMBUF          ; save it for the computation of byte 3
        TAD     @BUFPTR         ; do the same with the second word
        DCA     RAMBUF+1        ; ...
        SPD                     ; back to panel memory addressing

;   Store bytes 1 and 2 (they're easy) and calculate byte three.  Note that
; the SRAM will ignore the upper four bits when writing (there's no hardware
; there!) so there's no need to worry about masking them out first...
        JMS     @[RAMCDF]       ; select the correct SRAM "unit"
        MM3                     ; and enable the SRAM chips
        TAD     RAMBUF          ; store byte 1
        DCA     @RAMPTR         ; ...
        TAD     RAMBUF+1        ; and byte 2
        DCA     @RAMPTR         ; ...
        TAD     RAMBUF          ; byte 3 has the top four bits of word 1
        AND     [7400]          ; ...
        BSW                     ; ... in bits 8..11 of the byte
        CLL RTR                 ; ...
        DCA     RAMBUF          ; save that for a moment
        TAD     RAMBUF+1        ; and the top four bits of word 2
        AND     [7400]          ; ...
        CLL RTR                 ; in bits 4..7
        CLL RTR                 ; ...
        TAD     RAMBUF          ; ...
        DCA     @RAMPTR         ; ...
        MM2                     ; return to the default memory map
        CDF     1               ; and field
        ISZ     XFRCNT          ; have we done a whole page?
         JMP     PACK1          ; nope - keep going
        CLL                     ; be sure the LINK is cleared for success
        .POPJ                   ; all done
        .TITLE  Test RAM Disk Batteries


;   This routine tests the status of the RAM disk backup batteries.  The
; DS1221 doesn't have a status bit to give us the battery state directly, but
; it does have a clever hack to allow us to infer what we want to know.  If
; the batteries have failed, then the DS1221 will inhibit all chip select
; outputs on the _second_ memory cycle (but not the first!).  We can use this
; by 1) reading any location and saving its value, 2) writing any different
; value to the same location, and 3) reading it back again.  If the batteries
; are dead, the second cycle will be inhibited, and the value read in step 3
; will be the same as 1.  Of course, this presupposes that there's functional
; memory installed in the first place, if there isn't then this algorithm will
; erroneously report that the batteries are dead.
;
;   WARNING - because of the way the DS1221 battery test works, this function
; MUST be called before any other RAM disk accesses.
;
; NOTE: At this point, DF is 1, which selects RAM disk unit zero!
BATTST: STA                     ; ...
        DCA     BATTOK          ; assume batteries are OK for now
        LDAR                    ; and select SRAM bank zero
        MM3                     ; enable RAM disk access
        TAD     @[7777]         ; (1) read the last byte of this bank
        DCA     BATTMP          ; save it for a minute
        TAD     BATTMP          ; ...
        CIA                     ; make it negative
        DCA     @[7777]         ; (2) and write it back
        TAD     BATTMP          ; get the original data
        TAD     @[7777]         ; (3) add what should be the complement
        AND     [377]           ; ignore all but the bottom eight bits
        SZA CLA                 ; if it's not zero then the second cycle was
         DCA     BATTOK         ;  ... inhibited because the batteries are dead
        MM2                     ; back to the default memory map
```

```
        .POPJ                   ; ...

; Temporary storage for BATTST...
BATTMP: .BLOCK  1               ; ...
        .TITLE  Get Battery Status ROM Call


;   The Get Battery Status ROM call will return the status of the RAM disk
; lithium backup batteries.  As long as either battery has sufficient
; voltage, -1 will be return in the AC.  If both batteries have failed, then
; zero is returned.
;
;       PR0             / call the SBC6120 ROM firmware
;        0003           / get backup battery status function code
;                       / return with AC == -1 if batteries are OK
;
;   NOTE: Because of the way the DS1221 works, the battery status can only
; be tested after power up.  It isn't possible to monitor the battery status
; in real time!
GETBAT: CLA CLL                 ; this one's really easy!
        TAD     BATTOK          ; return the battery status in the AC
        .POPJ                   ; and that's it

        .PAGE
        .TITLE  Calculate RAM Disk Addresses


;   This routine will calculate the RAM disk bank number and the relative
; offset within that bank, corresponding to a disk page number passed in
; location DKPAGE.  The resulting bank number is simply loaded directly into
; the DAR via the LDAR IOT, and the offset is left in auto index location
; RAMPTR, where it can be used by the UNPACK and PACK routines.  If the page
; number passed is illegal (i.e. greater than the size of the selected RAM
; disk unit) then the link will be set when this routine returns.
SETDAR: CLA CLL                 ; make sure the link is in a known state
        TAD     RDPAGE          ; get the desired page
        TAD     RAMUSZ          ; compare it to the size of this unit
        SZL CLA                 ; is the page number legal?
         .POPJ                  ; no - return with the LINK set

;   Divide the page number by 21, the number of pages per bank, by repeated
; subtraction.  This is kind of crude, but it only has to iterate 127 times,
; worst case, so the performance hit isn't that bad.  We do have to be careful,
; though, because the largest legal page number is 2688, which is bigger than
; 2048.  That means we have to treat the whole AC as a 12 bit UNSIGNED value!
        DCA     RAMDAR          ; clear the disk address (quotient)
        TAD     RDPAGE          ; get the selected RAM disk page number
SETDA1: CLL                     ; make sure the link is clear before starting
        TAD     [-BANKSZ]       ; try to subtract another 21
        SNL                     ; did it fit?
         JMP     SETDA2         ; nope - we can stop now
        ISZ     RAMDAR          ; yes - increment the disk address
        JMP     SETDA1          ; and keep subtracting

;   We get here when we're done dividing, with the quotient in RAMDAR and the
; remainder in the AC.  To calculate the byte offset within a bank, we need
; to multiply the remainder by 192 (the number of bytes per 128 word page).
SETDA2: TAD     [BANKSZ]        ; restore the remainder
        BSW                     ; then multiply by 64
        DCA     RAMPTR          ; save offset*64 for a moment
        TAD     RAMPTR          ; ...
        CLL RAL                 ; then multiply by two again
        TAD     RAMPTR          ; 192*x = 128*x + 64*x
        TAD     [-1]            ; auto index registers pre-increment
        DCA     RAMPTR          ; that's the final offset

;   Set up the DAR with the bank number, from RAMDAR.  Remember that for the
; 128K chips, we must always set A17 to enable the alternate chip select!
        TAD     RAMUSZ          ; get the size of the selected unit
        TAD     [RAM128]        ; ...
        SNA CLA                 ; is this a 128K ram chip ?
         TAD     [32.]          ; yes - always set A17
```

```
        TAD     RAMDAR          ; get the quotient from the division
        LDAR                    ; and load the disk address register
        CLA CLL                 ; LDAR doesn't clear the AC!
        .POPJ                   ; and we're done
        .TITLE  Select RAM Disk Unit


;   This routine will set up the RAMCDF routine to select the desired RAM
; disk "unit".  The unit number, 0..3, should be passed in RDUNIT.  If the
; unit number given is illegal (i.e. greater than three) OR if there is no
; SRAM chip installed in the selected position, this routine will return
; with the link set.
RAMSEL: CLL CLA                 ; make sure the link is in a known state
        TAD     RDUNIT          ; get the desired unit number
        TAD     [-4]            ; see if the unit is legal
        SZL CLA                 ; it must be less than 4
         .POPJ                  ; no - return with the link set
        TAD     RDUNIT          ; restore the original unit
        CLL R3L                 ; and position it for a CDF instruction
        CLL RAL                 ; (the link is zero for a success return!)
        TAD     [CDF 0]         ; make a CDF to the corresponding field
        DCA     RAMCDF+1        ; and store that in the unit select routine

;   Now that we know the unit number is valid, verify that this chip is really
; installed by checking the RDSIZE array for a non-zero value.  As a side
; effect of this, we always leave the size of the currently selected unit in
; location RAMUSZ, where it's used by SETDAR to determine whether the page
; addressed is actually legal.  We always want to update RAMUSZ, even if the
; chip is not installed, because this will also cause SETDAR to fail if the
; caller ignores the our error return and attempts a read or write anyway.
        TAD     RDUNIT          ; get the unit number again
        TAD     [RDSIZE]        ; index into the RDSIZE array
        DCA     SIZPTR          ; ...
        TAD     @SIZPTR         ; to get the size of this chip
        CIA                     ; make it negative
        DCA     RAMUSZ          ; and save that for SETDAR
        TAD     RAMUSZ          ; ...
        CLL                     ; make sure the link is in a known state
        SNA CLA                 ; is this chip installed ?
         CML                    ; nope - give the error return
        .POPJ                   ; ...

;   This little routine can be called, via a JMS instruction (not a .PUSHJ!)
; to change the DF and select the last RAM disk unit set by a call to RAMSEL.
RAMCDF: 0                       ; call here via a JMS!
        NOP                     ; gets overwritten with a CDF instruction
        JMP     @RAMCDF         ; return...
        .TITLE  RAM Disk Diagnostics


;   This routine will do a simple test of the RAM disk array to determine
; whether each SRAM chip, from 0 to 3, is installed.  If a given chip is
; installed, then we do another simple test to determine whether it is a
; 512K or 128K device, and then update the RDSIZE array accordingly.
; Because of the way disk sectors are laid out, only the first 4032 bytes
; (21 * 192) of every 4Kb bank are actually used.  The last 64 bytes of each
; bank are available to us to use any way we like, including as a RAM test.
;
;   There's one nasty complication here - the pin that corresponds to A17 on
; the 512K SRAM chips is actually an alternate chip enable on the 128K chips.
; Worse, this alternate enable is active HIGH, which means that A17 must be
; one or 128K chips won't talk at all.  Fortunately, the pin that corresponds
; to A18 is a no connect on the 128K chips, so we can safely leave it zero.
; This explains the strange bank numbers selected in this test!
;
RDTEST: DCA     RDUNIT          ; start testing with unit zero
        DCA     RAMSIZ          ; clear the total RAM size
RDTES0: .PUSHJ  RAMSEL          ; and set up RAMCDF and SIZPTR

;   First test to see if this chip is even installed by writing alternating
; bit patterns to the last two locations and reading them back.  If that works,
; then there must be something there!
```

```
        TAD     [32.]           ; test bank 32 so that A17 will be set
        LDAR                    ; ...
        CLA                     ; (LDAR doesn't clear the AC!)
        JMS     RAMCDF          ; change the DF to select the unit
        MM3                     ; and enable the SRAM array
        TAD     [252]           ; write alternating bits to the last two bytes
        DCA     @[7776]         ; ...
        TAD     [125]           ; ...
        DCA     @[7777]         ; ...
        TAD     @[7776]         ; now read 'em back
        TAD     @[7777]         ; and add them up
        IAC                     ; the sum should be 377, so make it 400
        AND     [377]           ; and remember RAM disk is only 8 bits wide
        SZA CLA                 ; did it work??
         JMP     RDTES1         ; no - this chip doesn't exist

;   Some kind of SRAM chip is installed, and now we need to decide whether its
; 128K or 512K.  The 128K chips ignore A18, so one easy test is to select
; bank 96 (which, to a 128K chip is the same as bank 32), zero out a location
; that we just tested, and then go back to bank 32 to see if it changed.
        TAD     [96.]           ; select bank 96
        LDAR                    ; which doesn't exist in a 128K chip
        CLA                     ; (LDAR doesn't clear the AC!!)
        DCA     @[7777]         ; this location in bank 0 used to hold 125
        TAD     [32.]           ; back to bank 32
        LDAR                    ; ...
        CLA                     ; ...
        TAD     @[7777]         ; and see what we've got
        AND     [377]           ; remember RAM disk is only 8 bits wide
        SZA CLA                 ; if it's zero, then we have a 128K chip
         TAD     [RAM512-RAM128]; nope - this must be a full 512K SRAM!
        TAD     [RAM128]        ; only 128K, but better than nothing

; Store the chip size in RDSIZE and accumulate the total size...
RDTES1: MM2                     ; return to the default memory map
        CDF     1               ; and field
        DCA     @SIZPTR         ; store the size in RDSIZE[unit]
        TAD     @SIZPTR         ; ...
        SNA                     ; was there any chip here at all?
         JMP     RDTES2         ; no - we can skip this part
        SPA CLA                 ; KLUDGE - skip if this was a 128K chip
         TAD     [512.-128.]    ; add 512K to the total RAM size
        TAD     [128.]          ; add 128K  "  "    "    "   "
        TAD     RAMSIZ          ; ...
        DCA     RAMSIZ          ; ...

; On to the next unit, if there are any more left...
RDTES2: ISZ     RDUNIT          ; go on to the next unit
        TAD     RDUNIT          ; have we done all four ?
        TAD     [-4]            ; ???
        SZA CLA                 ; ???
         JMP    RDTES0          ; no - keep checking
        TAD     RAMSIZ          ; yes - return the total RAM size in the AC
        .POPJ                   ; and that's it

        .PAGE
        .TITLE  Get RAM Disk Size ROM Call


;   PR0 function 2 will return the size of a RAM disk chip, in 128 word pages,
; in the AC.  The AC should be loaded with the desired unit number, 0..3,
; before invoking PD0.  If no chip is installed in the selected unit, zero
; will be returned in the AC.  If the unit number is not in the range 0..3,
; then on return the LINK will be set to indicate an error.
;
; For example:
;       TAD     (unit   / load the desired RAM disk unit, 0..3
;       PR0             / call the ROM software
;        0002           / function code for Get RAM Disk Status
;                       / return the RAM disk size in the AC

;   It's tempting to use the RAMSEL routine here to save some steps, but be
```

```
; careful - RAMSEL will return with the LINK set (an error condition) if a
; valid unit number is selected but there is no SRAM chip installed there.
; That's not what we want for this ROM call, which should return an error only
; if the selected unit is > 3!
GETRDS: DCA     RDUNIT          ; save the unit number
        TAD     RDUNIT          ; and get it back
        CLL                     ; be sure the link is in a known state
        TAD     [-4]            ; is it a legal unit number ?
        SZL CLA                 ; skip if so
         .POPJ                  ; no - return with the LINK set and AC clear
        TAD     RDUNIT          ; one more time
        TAD     [RDSIZE]        ; index the RDSIZE array
        DCA     SIZPTR          ; ...
        TAD     @SIZPTR         ; get the size of this disk
        .POPJ                   ; and return it with the LINK cleared
        .TITLE  ATA Disk Support


;   BTS6120 supports any standard ATA hard disk connected to the SBC6120 IDE
; interface.  Nearly all hard disks with IDE interfaces are ATA; conversely,
; nearly all non-hard disk devices (e.g. CDROMs, ZIP drives, LS-120s, etc)
; with IDE interfaces are actually ATAPI and not ATA.  ATAPI requires a
; completely different protocol, which BTS6120 does not support, and BTS6120
; will simply ignore any ATAPI devices connected to the IDE interface.
; BTS6120 supports only a single physical drive, which must be set up as the
; IDE master, and any IDE slave device will be ignored.
;
;   Since BTS6120 does not support cylinder/head/sector (C/H/S) addressing,
; the hard disk used must support logical block addressing (LBA) instead.
; All modern IDE/ATA drives support LBA, as do most drives manufactured in the
; last five or six years, however some very old drives may not.  If BTS6120
; detects an ATA drive that does not support LBA it will display the message
; "IDE: Not supported" during startup and there after ignore the drive.
;
;   All IDE devices, regardless of vintage, transfer data in sixteen bit words
; and each sector on an ATA disk contains 512 bytes, or 256 sixteen bit words.
; When writing to the disk, BTS6120 converts twelve bit PDP-8 words to sixteen
; bits by adding four extra zero bits to the left and, when reading from the
; disk, BTS6120 converts sixteen bit words to twelve bits by simply discarding
; the most significant four bits.  No packing is done.  This conveniently
; means that each ATA sector holds 256 PDP-8 words, or exactly one OS/8 block.
; It also means that one quarter of the disk space is wasted, in this era of
; multi-gigabyte disks that hardly seems like an issue.
;
;   OS/8 handlers and the OS/8 file system use a single twelve bit word to hold
; block numbers, which means that OS/8 mass storage devices are limited to a
; maximum of 4096 blocks .  Using the BTS6120 non-packing scheme for storing
; data, 4096 PDP-8 blocks are exactly 2Mb.  Clearly, if a single OS/8 device
; corresponds to an entire hard disk then nearly all of the disk space would
; be wasted.  The normal solution is to partition the hard disk into many OS/8
; units, with each unit representing only a part of the entire disk.  Since
; OS/8 cannot support a single unit larger than 2Mb there isn't any point in
; allowing partitions to be larger than that, and since the smallest drives
; available today can hold hundreds if not thousands of 2Mb partitions, there
; isn't much point in allowing a partition to be smaller than that, either.
;
;   Because of this, BTS6120 supports only fixed size partitions of 2Mb each.
; This greatly simplifies the software since a twenty four bit disk sector
; number can now be calculated simply by concatenating a twelve bit partition
; number with the twelve bit OS/8 relative block number (RBN).  No "super
; block" with a partition table is needed to keep track of the sizes and
; positions of each partition, and the OS/8 handler is simplified since each
; disk partition is always the same size.  A twenty four bit sector address
; permits disks of up to 8Gb to be fully used, which seems more than enough
; for a PDP-8.
;
;   Once again, in BTS6120 the partition number simply refers to the most
; significant twelve bits of a twenty-four bit disk address, and the OS/8
; block number is the least significant twelve bits.  It's no more complicated
; than that!
;
;   The ID01 is the OS/8 handler for the SBC6120 IDE/ATA disk.  It supports
```

```
; eight units, IDA0 through IDA7, in a single page and may be assembled as
; either a system (IDSY) or non-system (IDNS) handler. The system handler
; version of the ID01 contains a secondary bootstrap that can be booted by
; the BTS6120 Boot command.  The ID01 is a simple handler that uses HD-6120
; PR0 instruction to invoke BTS6120  functions for low level IDE disk access
; and data transfer.
;
;   BTS6120 implements a partition map which defines the partition number
; corresponding to each OS/8 ID01 unit, and when an OS/8 program accesses an
; ID01 unit BTS6120 uses this table to determine the upper twelve bits of the
; LBA.  At power on or after a MR command, BTS6120 initializes this partition
; map so that unit 0 accesses partition 0, unit 1 accesses partition 1, and
; so on up through unit 7 and partition 7.  This mapping remains in effect
; until it is changed by either the PM command, or the Set IDE Disk Partition
; Mapping PR0 function.
;
;   The largest mass storage device supported by OS/8 is actually only 4095
; blocks, not 4096, and so the last block of every 2Mb partition is never
; used by OS/8.  This block can, however, be accessed via the Read/Write IDE
; disk PR0 function (section 6.5), and it can be used to store the name,
; creation date, and other information about that partition.  The OS/8 PART
; program uses this to allow partitions to be mounted on ID01 logical units
; by name rather than partition number.  Named partitions are strictly a
; function of the OS/8 PART program and BTS6120 knows nothing about them.
        .TITLE  IDE Disk Interface


;   In the SBC6120, the IDE interface is implemented via a standard 8255 PPI,
; which gives us 24 bits of general purpose parallel I/O.  Port A is connected
; the high byte (DD8..DD15) of the IDE data bus and port B is connected to
; the low byte (DD0..DD7).  Port C supplies the IDE control signals as follow:
;
;   C0..C2 -> DA0 .. 2 (i.e. device address select)
;   C.3*   -> DIOR L (I/O read)
;   C.4*   -> DIOW L (I/O write)
;   C.5*   -> RESET L
;   C.6*   -> CS1Fx L (chip select for the 1Fx register space)
;   C.7*   -> CS3Fx L ( "     "     "   "  3Fx     "       "   )
;
; * These active low signals (CS1Fx, CS3Fx, DIOR, and DIOW) are inverted in
; the hardware so that writing a 1 bit to the register asserts the signal.
;
;   One nice feature of the 8255 is that it allows bits in port C to be
; individually set or reset simply by writing the correct command word to the
; control register - it's not necessary to read the port, do an AND or OR,
; and write it back.  We can use this feature to easily toggle the DIOR and
; DIOW lines with a single PWCR IOT.
IDEINP=222      ; set ports A and B as inputs, C as output
IDEOUT=200      ; set ports A and B (and C too) as outputs
SETDRD=007      ; assert DIOR L (PC.3) in the IDE interface
CLRDRD=006      ; clear    "  "  "    "   "  "   "    "
SETDWR=011      ; assert DIOW L (PC.4) in the IDE interface
CLRDWR=010      ; clear    "  "  "    "   "  "   "    "
SETDRE=013      ; assert DRESET L (PC.5) in the IDE interface
CLRDRE=012      ; clear     "   "    "    "  "   "    "    "

; Standard IDE registers...
;   (Note that these are five bit addresses that include the two IDE CS bits,
; CS3Fx (AC4) and CS1Fx (AC5).  The three IDE register address bits, DA2..DA0
; correspond to AC9..AC11.
CS1FX=100       ; PC.6 selects the 1Fx register space
CS3FX=200       ; PC.7    "     "  3Fx   "   "    "
REGDAT=CS1FX+0  ; data  (R/W)
REGERR=CS1FX+1  ; error (R/O)
REGCNT=CS1FX+2  ; sector count (R;W)
REGLB0=CS1FX+3  ; LBA byte 0 (or sector number) R/W
REGLB1=CS1FX+4  ; LBA byte 1 (or cylinder low) R/W
REGLB2=CS1FX+5  ; LBA byte 2 (or cylinder high) R/W
REGLB3=CS1FX+6  ; LBA byte 3 (or device/head) R/W
REGSTS=CS1FX+7  ; status (R/O)
REGCMD=CS1FX+7  ; command (W/O)
```

```
; IDE status register (REGSTS) bits...
STSBSY=0200     ; busy
STSRDY=0100     ; device ready
STSDF= 0040     ; device fault
STSDSC=0020     ; device seek complete
STSDRQ=0010     ; data request
STSCOR=0004     ; corrected data flag
STSERR=0001     ; error detected

; IDE command codes (or at least the ones we use!)...
CMDEDD=220      ; execute device diagnostic
CMDIDD=354      ; identify device
CMDRDS=040      ; read sectors with retry
CMDWRS=060      ; write sectors with retry
CMDSUP=341      ; spin up
CMDSDN=340      ; spin down
        .TITLE  Initialize IDE Drive and Interface


;   This routine will initialize the IDE interface by configuring the 8255
; PPI and then asserting the IDE RESET signal to the drive.  It then selects
; the master drive and waits for it to become ready, after which it returns.
; If there is no drive attached, or if the hardware is broken, then we'll time
; out after approximately 30 seconds of waiting for the drive to signal a
; ready status.
;
;   Normally this routine will return with the AC and LINK both cleared, but
; if the drive reports an error then on return the LINK will be set and the
; drive's error status will be in the AC.  In the case of a timeout, the
; LINK will be set and the AC will be -1 on return.
IDEINI: CLA                     ; ...
        DCA     DKSIZE          ; zero means no disk is installed
        TAD     [IDEINP]        ; PPI ports A and B are input and C is output
        PWCR                    ; ...
        PWPC                    ; clear all port C control lines
        TAD     [SETDRE]        ; set RESET L
        PWCR                    ; ...
        TAD     [-10]           ; according to the ATA specification,
        IAC                     ;  ... we must leave RESET asserted for at
        SZA                     ;  ... least 25 microseconds
         JMP     .-2            ;
        TAD     [CLRDRE]        ; deassert RESET L
        PWCR                    ; ...
        TAD     [340]           ; select the master drive, 512 byte sectors,
        JMS     @[IDEWRR]       ;  ... and logical block addressing (LBA) mode
         REGLB3                 ; ...
        JMP     @[WREADY]       ; wait for the drive ready and return
        .TITLE  Identify IDE/ATA Device


;   This routine will execute the ATA IDENTIFY DEVICE command and store the
; first 256 bytes of the result, in byte mode, in the panel memory buffer at
; DSKBUF.  One thing to keep in mind is that ATAPI devices (e.g. CDROMs, ZIP
; disks, etc) ignore this command completely and respond to the ATAPI IDENTIFY
; PACKET DEVICE command instead.  This means that if there are any ATAPI
; devices attached we'll never see them, which is fine since we don't
; understand how to talk to ATAPI devices anyway!
;
;   The drive's response to IDENTIFY DEVICE will be 256 words of sixteen bit
; data full of device specific data - model number, manufacturer, serial
; number, drive geometry, maximum size, access time, and tons of other cool
; stuff.  The RDIBUF routine would pack this sixteen bit data into twelve bit
; words by throwing away the upper four bits of each word, but that doesn't
; make sense in this case since we'd be destroying most of the useful
; information.  Instead, this routine reads the data in an unpacked format and
; stores one eight bit byte per PDP-8 word.
;
;   Unfortunately this would mean that we need a 512 word buffer to store the
; response, which is too big for our DSKBUF in panel memory.  We're in luck,
; however, because of the 256 words (512 bytes) returned by this command the
; ATA specification only defines the first 128 - the remaining half of the
; data is "vendor specific" and undefined.  This routine simply throws this
```

```
; part away, and only the first 128 words (256 bytes) of the drive's response
; are actually returned in the buffer.
;
;   Like all the disk I/O routines, in the case of an error the LINK will
; be set and the contents of the drive's error register returned in the AC.
DISKID: .PUSHJ  @[WREADY]       ; (just in case the drive is busy now)
        SZL                     ; any errors?
         .POPJ                  ; yes - we can go home early!
        TAD     [CMDIDD]        ; send the ATA identify device command
        JMS     @[IDEWRR]       ; by writing it to the command register
         REGCMD                 ; ...
        .PUSHJ  @[WDRQ]         ; the drive should ask to transfer data next
        SZL                     ; any errors?
         .POPJ                  ; yes - just give up

; Get ready to ready to transfer data from the drive to our buffer...
        TAD     [DSKBUF-1]      ; setup BUFPTR to point to DSKBUF
        DCA     BUFPTR          ; ...
        TAD     [-128.]         ; transfer 128 words this time
        DCA     XFRCNT          ; ...
        TAD     [IDEINP]        ; set PPI ports A and B to input mode
        PWCR                    ; ...
        TAD     [REGDAT]        ; make sure the IDE data register is selected
        PWPC                    ; ...

;   Read 256 bytes into the caller's buffer, one byte per word.  Big endian
; ordering (i.e. high byte first) is defined by the ATA specification to give
; the correct character order for ASCII strings in the device data (e.g. model
; number, serial number, manufacturer, etc).
IDDEV1: TAD     [SETDRD]        ; assert DIOR
        PWCR                    ; ...
        PRPA                    ; read port A (the high byte) first
        AND     [377]           ; only eight bits are valid
        DCA     @BUFPTR         ; and store it in the buffer
        PRPB                    ; then read port B (the low byte)
        AND     [377]           ; ...
        DCA     @BUFPTR         ; ...
        TAD     [CLRDRD]        ; deassert DIOR
        PWCR                    ; ...
        ISZ     XFRCNT          ; have we done all 256 bytes?
         JMP     IDDEV1         ; nope - keep reading

;   We've read our 256 bytes, but the drive still has another 256 more waiting
; in the buffer.  We need to read those and throw them away...
        TAD     [-128.]         ; we still need to read 128 more words
        DCA     XFRCNT          ; ...
IDDEV2: TAD     [SETDRD]        ; assert DIOR
        PWCR                    ; ...
        NOP                     ; make sure the DIOR pulse is wide enough
        NOP                     ; ...
        TAD     [CLRDRD]        ; and then clear DIOR
        PWCR                    ; ...
        ISZ     XFRCNT          ; have we done all 128?
         JMP     IDDEV2         ; nope - keep reading

;   Drives report the total number of LBA addressable sectors in words
; 60 and 61.  Sectors are 512 bytes, so simply dividing this value by 2048
; gives us the total drive size in Mb.  This code patches together twelve
; bits out of the middle of this doubleword, after throwing away the least
; significant 11 bits to divide by 2048.  This allows us to determine the
; size of drives up to 4Gb in a single 12 bit word.
        TAD     @[DSKBUF+170]   ; get the high byte of the low word
        RAR                     ; throw away the 3 least significant
        RTR                     ; ...
        AND     [37]            ; keep just 5 bits from this byte
        DCA     DKSIZE          ; save it for a minute
        TAD     @[DSKBUF+173]   ; get the low byte of the high word
        RTL                     ; left justify the seven MSBs of it
        RTL                     ; ...
        RAL                     ; ...
        AND     [7740]          ; ...
        TAD     DKSIZE          ; put together all twelve bits
```

```
        DCA     DKSIZE          ; ...

; All done - return success...
        CLA CLL                 ; return with the AC and LINK clear
        .POPJ                   ; ...
        .TITLE  Get IDE Disk Size ROM Call


;   The get IDE disk size call will return the size of the attached IDE/ATA
; disk, in megabytes.  This call never fails - if no disk is attached it
; simply returns zero...
;
;CALL:
;       PR0             / call SBC6120 ROM firmware
;        5              / subfunction for get disk size
;       <return here with disk size, in megabytes, in AC>
;
GETDKS: CLA CLL                 ; ignore anything in the AC
        TAD     DKSIZE          ; and return the disk size
        .POPJ                   ; that's all there is to it!
        .TITLE  IDE Disk Primary Bootstrap


;   This routine will read block zero from IDE disk partition zero into page
; zero of field zero of main memory.  The next step in the usual boot sequence
; would be to start the secondary bootstrap, but that's up to the caller...
IDBOOT: STA                     ; point the buffer to page 0
        DCA     BUFPTR          ; ...
        TAD     [CDF 0]         ; of field zero
        DCA     @[BUFCDF+1]     ; ...
        DCA     BUFPNL          ; of main memory
        DCA     DKPART          ; read IDE disk partition zero
        DCA     DKRBN           ; block zero
        TAD     [-128.]         ; we only need the first 1/2 of the block
        JMP     @[DISKRD]       ; ...

        .PAGE
        .TITLE  Read and Write IDE Sectors


;   This routine will read a single sector from the attached IDE drive.
; The caller should set up DKPART and DKRBN with the disk partition and
; sector number, and BUFPTR, BUFCDF and BUFPNL with the address of a
; buffer in 6120 memory.  If any errors are encountered, this routine will
; return with the LINK set and the drive's error status in the AC...
DISKRD: DCA     BUFSIZ          ; save the buffer size
        .PUSHJ  WREADY          ; wait for the drive to become ready
        SZL                     ; any errors detected??
         .POPJ                  ; yes - quit now
        .PUSHJ  SETLBA          ; set up the disk's LBA registers
        TAD     [CMDRDS]        ; read sector with retry command
        JMS     @[IDEWRR]       ; write that to the command register
         REGCMD                 ; ...
        .PUSHJ  WDRQ            ; now wait for the drive to finish
        SZL                     ; any errors detected?
         .POPJ                  ; yes - quit now
        TAD     BUFSIZ          ; no - transfer data
        JMP     @[RDIBUF]       ;  ... from the sector buffer to memory


;   This routine will write a single sector to the attached IDE drive.  Except
; for the direction of data transfer, it's basically the same as DISKRD,
; including all parameters and error returns.
DISKWR: DCA     BUFSIZ          ; save the caller's record size
        .PUSHJ  WREADY          ; wait for the drive to become ready
        SZL                     ; did we encounter an error ?
         .POPJ                  ; yes - just give up now
        .PUSHJ  SETLBA          ; set up the disk address registers
        TAD     [CMDWRS]        ; write sector with retry command
        JMS     @[IDEWRR]       ; write that to the command register
         REGCMD                 ; ...
        .PUSHJ  WDRQ            ; wait for the drive to request data
```

```
        SZL                     ; did the drive detect an error instead?
         .POPJ                  ; yes - just give up
        TAD     BUFSIZ          ; nope - transfer the data
        .PUSHJ  @[WRIBUF]       ;  ... to the sector buffer from memory

;   There's a subtle difference in the order of operations between reading and
; writing.  In the case of writing, we send the WRITE SECTOR command to the
; drive, transfer our data to the sector buffer, and only then does the
; drive actually go out and access the disk.  This means we have to wait
; one more time for the drive to actually finish writing, because only then
; can we know whether it actually worked or not!
        JMP     WREADY          ; wait for the drive to finish writing
        .TITLE  Spin Up and Spin Down IDE Drive


;   This routine will send a spin up command to the IDE drive and then wait
; for it to finish.  This command will take a fairly long time under normal
; conditions.  Worse, since this is frequently the first command we send to
; a drive, if there's no drive attached at all we'll have to wait for it
; to time out.  If any errors are encountered then the LINK will be set on
; return and the contents of the drive's error register will be in the AC.
SPINUP: CLA                     ; ...
        TAD     [CMDSUP]        ; send the spin up command to the drive
        JMS     @[IDEWRR]       ; by writing it to the command register
         REGCMD                 ; ...
        JMP     WREADY          ; wait for the drive to become ready


;   This routine will send a spin down command.  Drives are not required by
; the standard to implement this command, so there's no guarantee that any
; thing will actually happen!
SPINDN: CLA                     ; ...
        TAD     [CMDSDN]        ; send the spin down command to the drive
        JMS     @[IDEWRR]       ; ...
         REGCMD                 ; ...
        JMP     WREADY          ; and wait for it to finish
        .TITLE  Setup IDE Unit, LBA and Sector Count Registers


;   This routine will set up the IDE logical block address (LBA) registers
; according to the current disk address in locations DKPART and DKRBN.  On IDE
; drives the sector number is selected via the cylinder and sector registers in
; the register file, but in the case of LBA mode these registers simply form a
; 24 bit linear sector number.  In this software the disk partition number, in
; DKPART, gives the upper twelve bits of the address and the current sector
; number, in DKRBN, gives the lower twelve bits.
;
;   This routine does not detect any error conditions...
SETLBA: CLA                     ; just in case!
        TAD     DKRBN           ; get the lower 12 bits of of the LBA
        JMS     @[IDEWRR]       ; and write the lowest 8 bits to LBA0
         REGLB0                 ;  (the upper 4 bits are ignored)
        TAD     DKRBN           ; now get the upper 4 bits of the sector number
        BSW                     ; shift right eight bits
        RTR                     ; ...
        AND     [17]            ; get rid of the extra junk
        DCA     LBATMP          ; ...
        TAD     DKPART          ; get the disk partition number
        RTL                     ; shift them left four bits
        RTL                     ; ...
        AND     [360]           ; and isolate just four bits of that
        TAD     LBATMP          ; and build the middle byte of the LBA
        JMS     @[IDEWRR]       ; set that register next
         REGLB1                 ; ...
        TAD     DKPART          ; get the partition one more time
        RTR                     ; shift it right four more bits
        RTR                     ; ...
        JMS     @[IDEWRR]       ; to make the upper byte of the 24 bit LBA
         REGLB2                 ; ...

;   Note that the final four bits of the LBA are in LBA3 (the head and drive
; select register).  Since we can only support 24 bit LBAs, these are unused.
```

```
; The IDEINI routine initializes them to zero at the same time it selects the
; master drive, and we never change 'em after that.  At the same time, IDEINI
; also selects LBA addressing mode (which is obviously very important to us!)
; and 512 byte sectors.
        TAD     [340]           ; select the master drive, 512 byte sectors,
        JMS     @[IDEWRR]       ;  ... and logical block addressing (LBA) mode
         REGLB3                 ; ...

; Always load the sector count register with one...
        NL0001                  ; write 1
        JMS     @[IDEWRR]       ;  ...
         REGCNT                 ;  to the sector count register
        .POPJ                   ; that's all we have to do

; Temporary storage for SETLBA...
LBATMP: .BLOCK  1
        .TITLE  Wait for IDE Drive Ready


;   This routine tests for the DRIVE READY bit set in the status register and
; at the same time for the DRIVE BUSY bit to be clear.  READY set means that
; the drive has power and is spinning, and BUSY clear means that it isn't
; currently executing a command.  The combination of these two conditions means
; that the drive is ready to accept another command.   Normally this routine
; will return with both the AC and the LINK cleared, however if the drive sets
; the ERROR bit in its status register then it will return with the LINK set
; and the contents of the drive's error register in the AC.
;
;   If there is no drive connected, or if the drive fails for some reason,
; then there is the danger that this routine will hang forever.  To avoid
; that it also implements a simple timeout, and if the drive doesn't become
; ready within a certain period of time it will return with the LINK set and
; the AC equal to -1.  If the system has just been powered up, then we'll
; have to wait for the drive to spin up before it becomes ready, and that can
; take a fairly long time.  To be safe, the timeout currently stands at a
; full 30 seconds!
WREADY: TAD     [7550]          ; initialize the outer timeout counter
        DCA     RDYTMO+1        ; ...
        DCA     RDYTMO          ; and the inner counter is always cleared
WREAD1: JMS     @[IDERDR]       ; go read the status register
         REGSTS                 ; (register to read)
        CLL RAR                 ; test the error bit first (AC11)
        SZL                     ; ???
         JMP     DRVERR         ; give up now if the drive reports an error
        RAL                     ; restore the original status
        AND     [STSBSY+STSRDY] ; test both the READY and BUSY bits
        TAD     [-STSRDY]       ; is READY set and BUSY clear?
        CML                     ; (the last TAD will have set the link!)
        SNA CLA                 ; ???
         .POPJ                  ; yes - return now with the AC and LINK clear
        ISZ     RDYTMO          ; increment the inner timeout counter
         JMP     WREAD1         ; no overflow yet
        ISZ     RDYTMO+1        ; when the inner counter overflows, increment
         JMP     WREAD1         ;  ... the outer counter too

; Here in the case of a drive time out...
        CLA CLL CML CMA         ; return with AC = -1 and the LINK set
        .POPJ                   ; ...

; Temporary storage for WREADY...
RDYTMO: .BLOCK  2               ; a double word time out counter
        .TITLE  Wait for IDE Data Request


;   This routine will wait for the DRQ bit to set in the drive status register.
; This bit true when the drive is ready to load or unload its sector buffer,
; and normally a call to this routine will be immediately followed by a call
; to ether RDIBUF or WRIBUF.  Normally this routine will return with both the
; LINK and the AC cleared, however if the drive sets its error bit then the
; LINK will be 1 on return and the drive's error status will be in the AC.
;
;   WARNING - unlike WREADY, this routine does not have a timeout!
```

```
WDRQ:   JMS     @[IDERDR]       ; read the drive status register
         REGSTS                 ; ...
        CLL RAR                 ; test the error bit (AC11)
        SZL                     ; is it set?
         JMP     DRVERR         ; yes - give the error return
        RAL                     ; no - restore the original status value
        AND     [STSBSY+STSDRQ] ; and test the BUSY and DRQ flags
        TAD     [-STSDRQ]       ; wait for BUSY clear and DRQ set
        CML                     ; (the last TAD will have set the link!)
        SZA CLA                 ; well?
         JMP     WDRQ           ; nope - keep waiting
        .POPJ                   ; yes - return with AC and LINK cleared!

;   We get here if the drive sets the error flag in the status register.  In
; this case we return with the link bit set and the contents of the drive
; error register in the AC.
DRVERR: JMS     @[IDERDR]       ; read the drive error register
         REGERR                 ; ...
        STL                     ; and be sure the link is set
        .POPJ                   ; ...

        .PAGE
        .TITLE  Write IDE Sector Buffer


;   This routine will write PDP-8 twelve bit words to the IDE drive's sixteen
; bit data (sector) buffer.  IDE drives naturally transfer data in sixteen bit
; words, and we simply store each twelve bit word zero extended.  This wastes
; 25% of the drive's capacity, but in these days of multiple gigabyte disks,
; that hardly seems important.  This also means that 256 PDP-8 words exactly
; fill one IDE sector, which is very convenient for OS/8!
;
;   The caller is expected to set up BUFPTR, BUFCDF and BUFPNL to point to the
; buffer in 6120 memory.  The negative of the buffer size should be passed in
; the AC, however we must always write exactly 256 words to the drive regard-
; less of the buffer size.  If the buffer is smaller than that, then the last
; word is simply repeated until we've filled the entire sector.  This is
; necessary for OS/8 handler "half block" writes.
;
;   This routine does not wait for the drive to set DRQ, nor does it check the
; drive's status for errors.  Those are both up to the caller.
WRIBUF: DCA     BUFSIZ          ; save the actual buffer size
        TAD     [-256.]         ; but always transfer 256 words, regardless
        DCA     XFRCNT          ; ...
        TAD     [IDEOUT]        ; and set ports A and B to output mode
        PWCR                    ; ...
        TAD     [REGDAT]        ; make sure the IDE data register is addressed
        PWPC                    ; ...
        JMS     @[BUFCDF]       ; change to the buffer's field

; Transfer 256 twelve bit words into 256 sixteen bit words...
WRIBU1: TAD     @BUFPTR         ; and get the next data word
        DCA     BUFTMP          ; save it temporarily
        TAD     BUFTMP          ; ...
        PWPB                    ; write the lowest 8 bits to port B
        TAD     BUFTMP          ; then get the upper four bits
        BSW                     ; ...
        RTR                     ; ...
        AND     [17]            ; ensure that the extra bits are zero
        PWPA                    ; and write the upper four bits to port A
        TAD     [SETDWR]        ; assert DIOW
        PWCR                    ; ...
        TAD     [CLRDWR]        ; and then clear it
        PWCR                    ; ...
        ISZ     XFRCNT          ; have we done 256 words??
         SKP                    ; no - keep going
          JMP     WRIBU3        ; yes - always stop now
        ISZ     BUFSIZ          ; have we filled the buffer ?
         JMP     WRIBU1         ; nope - keep copying

;   Here when we've emptied the 6120 buffer, but if we haven't done 256 words
; we have to keep going until we've filled the drive's sector buffer.  All we
```

```
; need to do is to keep asserting DIOW, which simply repeats the last word
; written!
WRIBU2: TAD     [SETDWR]        ; assert DIOW
        PWCR                    ; ...
        TAD     [CLRDWR]        ; and deassert DIOW
        PWCR                    ; ...
        ISZ     XFRCNT          ; have we finished the sector?
         JMP     WRIBU2         ; nope

;   Restore the PPI ports to input mode and return.  Note that some disk
; I/O routines JMP to WRIBUF as the last step, so it's important that we
; always return with the AC and LINK cleared to indicate success.
WRIBU3: CDF     1               ; return to our field
        SPD                     ; and to panel memory
        TAD     [IDEINP]        ; reset ports A and B to input
        PWCR                    ; ...
        CLA CLL                 ; return success
        .POPJ                   ; all done here
        .TITLE  Read IDE Sector Buffer


;   This routine will read sixteen bit words from the IDE drive's sector
; buffer and store them in twelve bit PDP-8 memory words.  Data is converted
; from sixteen to twelve bits by the simple expedient of discarding the upper
; four bits of each word - it can't get much easier than that!
;
;   The caller is expected to set up BUFPTR, BUFCDF and BUFPNL to point to the
; buffer in 6120 memory.  The negative of the buffer size should be passed in
; the AC.  This is the number of words that will be stored in the buffer,
; however we'll always read exactly 256 words from the drive regardless of
; the buffer size.  If the buffer is smaller than this then the extra words
; are simply discarded.  This is necessary for OS/8 handler "half block" reads.
;
;   Like WRIBUF, this routine does not wait for the drive to set DRQ, nor does
; it check the drive's status for errors.  Those are both up to the caller.
RDIBUF: DCA     BUFSIZ          ; save the actual buffer size
        TAD     [-256.]         ; but always transfer 256 words, regardless
        DCA     XFRCNT          ; ...
        TAD     [IDEINP]        ; and set ports A and B to input mode
        PWCR                    ; ...
        TAD     [REGDAT]        ; make sure the IDE data register is addressed
        PWPC                    ; ...
        JMS     @[BUFCDF]       ; change to the buffer's field

; Transfer 256 twelve bit words...
RDIBU1: TAD     [SETDRD]        ; assert DIOR
        PWCR                    ; ...
        PRPB                    ; capture the lower order byte
        AND     [377]           ; remove any junk bits, just in case
        DCA     BUFTMP          ; and save that for a minute
        PRPA                    ; then capture the high byte
        AND     [17]            ; we only want four bits from that
        BSW                     ; shift it left eight bits
        CLL RTL                 ; ...
        TAD     BUFTMP          ; assemble a complete twelve bit word
        DCA     @BUFPTR         ; and store it in the buffer
        TAD     [CLRDRD]        ; finally we can deassert DIOR
        PWCR                    ; ...
        ISZ     XFRCNT          ; have we done 256 words??
         SKP                    ; no - keep going
          JMP     RDIBU3        ; yes - always stop now
        ISZ     BUFSIZ          ; have we filled the buffer ?
         JMP     RDIBU1         ; nope - keep copying

;   Here when we've filled the 6120 buffer, but if we haven't done 256 words
; we have to keep going until we've emptied the drive's sector buffer too.
; All we need to do is to keep asserting DIOR - there's no need to actually
; capture the data!
RDIBU2: TAD     [SETDRD]        ; assert DIOR
        PWCR                    ; ...
        TAD     [CLRDRD]        ; and deassert DIOR
        PWCR                    ; ...
```

```
        ISZ     XFRCNT          ; have we finished the sector?
         JMP     RDIBU2         ; nope

;   Restore the ROM field and memory space and return.  Note that some disk
; I/O routines JMP to RDIBUF as the last step, so it's important that we
; always return with the AC and LINK cleared to indicate success.
RDIBU3: CDF     1               ; ...
        SPD                     ; ...
        CLA CLL                 ; always return success
        .POPJ                   ; all done here

; Temporary storage for RDIBUF and WRIBUF...
BUFTMP: .BLOCK  1               ; temporary for packing and unpacking
        .TITLE  Initialize Disk Partition Map


;   This routine will initialize the disk partition map so that unit 0
; maps to partition 0, unit 1 maps to partition 1, etc...  This is the
; default partition mapping used after a power on and remains in effect
; until changed by an OS/8 program with the "Set Partition Mapping" PR0
; subfunction.
INIPMP: CLA CLL                 ; just in case...
        TAD     [PARMAP-1]      ; set up an auto index register
        DCA     XX1             ;  ... to address the partition map
        DCA     DKPART          ; count partition/unit numbers here
INIPM1: TAD     DKPART          ; get the current partition/unit
        DCA     @XX1            ; and set the next entry in the map
        TAD     DKPART          ; see how many we've done
        TAD     [-10]           ; have we done all eight?
        SZL CLA                 ; skip if not
         .POPJ                  ; yes - we can quit now
        ISZ     DKPART          ; nope - do the next one
        JMP     INIPM1          ; ...

        .PAGE
        .TITLE  Get/Set Disk Partition Map ROM Call


;   This routine handles the "Set Disk Partition Mapping" (6) PR0 subfunction,
; which simply sets the partition number for the specified OS/8 unit.  This
; change takes effect immediately, so if you've booted from the IDE disk
; you'll want to be a little careful about remapping the system partition!
; This function returns with the LINK set if an error occurs, currently the
; only failure that can happen is if the unit number is .GT. 7.  Note that
; no range checking is done on the partition number to ensure that it fits
; within the disk size - if it doesn't we'll simply get I/O errors when OS/8
; attempts to access that partition.
;
;CALL:
;       TAD     (part   / load the partition number into the AC
;       PR0             / invoke the ROM monitor
;        6              / subfunction for Set disk partition
;        <unit>         / OS/8 unit to be changed, 0..7
;       <return>        / LINK set if unit .GT. 7
;
SETPMP: DCA     DKPART          ; save the partition number for a minute
        .PUSHJ  @[GETARG]       ; and get the unit number
        DCA     DKUNIT          ; ...
        TAD     DKUNIT          ; ...
        CLL                     ; be sure the link is in a known state
        TAD     [-10]           ; see if the unit number is legal
        SZL CLA                 ; the link will be set if it isn't
         .POPJ                  ; take the error return w/o changing anything
        TAD     DKUNIT          ; construct an index to the partition map
        TAD     [PARMAP-1]      ; ...
        DCA     XX1             ; ...
        TAD     DKPART          ; then get the desired partition number
        DCA     @XX1            ; and change it
        .POPJ                   ; return with the LINK and AC both clear


;   This routine handles the "Get Disk Partition Mapping" (7) PR0 subfunction,
```

```
; which simply returns the partition number currently associated with a
; specific OS/8 unit.  The only way it can fail is if the unit number is
; greater than 7!
;
;CALL:
;       PR0             / invoke the ROM monitor
;        7              / subfunction for get disk partition
;        <unit>         / OS/8 unit to be changed, 0..7
;       <return>        / with partition number in the AC
;
GETPMP: .PUSHJ  @[GETARG]       ; and get the unit number
        DCA     DKUNIT          ; ...
        CLL                     ; be sure the link is in a known state
        TAD     [-10]           ; see if the unit number is legal
        TAD     DKUNIT          ; ...
        SZL CLA                 ; the link will be set if it isn't
         .POPJ                  ; take the error return
        TAD     DKUNIT          ; construct an index to the partition map
        TAD     [PARMAP-1]      ; ...
        DCA     XX1             ; ...
        TAD     @XX1            ; and get the current partition
        .POPJ                   ; return with partition in the AC and LINK=0
        .TITLE  IDE Disk Read/Write ROM Call


; The calling sequence for the PR0 IDE disk R/W function is:
;
;       PR0
;        0004           / panel function code for IDE disk I/O
;        <arg1>         / R/W bit, page count, buffer field and unit
;        <arg2>         / buffer address
;        <arg3>         / starting block number
;       <return>        / if any errors occur, the LINK will be set and the
;                       /   the drive's error register are in the AC
;
;   Except for the function code, the use of block numbers instead of page
; numbers, and the error codes, this calling sequence is identical to the
; RAM disk I/O PR0 subfunction!
;
DISKRW: .PUSHJ  @[SETBUF]       ; set up MUUO, BUFPTR, BUFCDF and RWCNT
        .PUSHJ  @[GETARG]       ; and lastly get the disk block
        DCA     DKRBN           ; ...

;   See if there really is a hard disk attached.  If not, then immediately
; take the error return with the AC set to -1.
        TAD     DKSIZE          ; if there is a disk attached
        SZA CLA                 ; then DKSIZE will be non-zero
         JMP     DKRW0          ; it is - it's safe to proceed
        CLA CLL CML CMA         ; no disk - return LINK = 1 and AC = -1
        .POPJ                   ; and quit now

;   The unit number is really just an index into the partition table and,
; since it's limited to three bits and eight units are supported, there's
; no need to range check it!
DKRW0:  TAD     MUUO            ; get the unit number
        AND     [7]             ; ...
        TAD     [PARMAP-1]      ; create an index to the partition table
        DCA     XX1             ; ...
        TAD     @XX1            ; get the actual partition number
        DCA     DKPART          ;  ... that's mapped to this unit

;   Set up a pointer to the I/O routine.  All of the rest of this code is
; independent of the direction of data flow...
        TAD     MUUO            ; get the function code
        SMA CLA                 ; should we read (0) or write (1) ?
         TAD     [DISKRD-DISKWR];  ... read
        TAD     [DISKWR]        ;  ... write
        DCA     DISKIO          ; save the address of the routine

;   We must take a minute out to share a word about pages vs blocks.  An OS/8
; handler call specifies the size of the data to be read or written in pages,
; which are 128 words or exactly 1/2 of a 256 word disk block.  This raises
```

```
; the unfortunate possibility that a program could ask to transfer an odd
; number of pages, which would mean that we'd need to read or write half a
; block!  We can't ignore this problem because it really does happen and there
; really are OS/8 programs that attempt to transfer an odd number of pages.
;
;   This is primarily an issue for reading, because if an odd number of pages
; are to be read we must be very careful to stop copying data to memory after
; 128 words.  If we don't, a page of memory will be corrupted by being over
; written with the second half of the last disk block!  It's also permitted in
; OS/8 to write an odd number of pages, but since many OS/8 mass storage
; devices have 256 word sectors it isn't always possible to write half a
; block.  In this case it's undefined what gets written to the last half of
; the final block - it could be zeros, random garbage, or anything else.

; This loop reads or writes pages 'till we've done all we're supposed to...
DKRW1:  ISZ     RWCNT           ; is there an odd page left over?
         SKP                    ; nope - it's safe to do a full block
          JMP     DKRW2         ; yes - go transfer a half block and quit
        TAD     [-256.]         ; transfer two pages this time
        .PUSHJ  @DISKIO         ; either read or write
        SZL                     ; were there any errors?
         .POPJ                  ; yes - just abort the transfer now
        ISZ     DKRBN           ; increment the block number for next time
         NOP                    ;  (this should never happen, but...)
        ISZ     RWCNT           ; are there more pages left to do ?
         JMP     DKRW1          ; yup - keep going
        .POPJ                   ; all done - return AC = LINK = 0

; Here to transfer one, final, half block...
DKRW2:  TAD     [-128.]         ; only do a single page this time
        JMP     @DISKIO         ; transfer it and we're done

; Local storage for DISKRW...
DISKIO: .BLOCK  1               ; gets a pointer to either DISKRD or DISKWR
        .TITLE  Write IDE Register


;   This routine will write an eight bit value to any IDE drive register,
; except the data regsister, by toggling all the appropriate PPI port lines.
; The address of the register, which should include the CS1Fx and CS3Fx bits,
; is passed in line and the byte to be written is passed in the AC.  Note that
; all IDE registers, with the exception of the data register, are eight bits
; wide so there's never a need to worry about the upper byte!
;
;CALL:
;       TAD     [value] ; eight bit value to write to the IDE register
;       JMS     IDEWRR
;        xxxx           ; IDE register number, plus CS1Fx and CS3Fx bits
;       <always return here, with AC cleared>
;
IDEWRR: 0                       ; CALL HERE WITH A JMS!!!
        DCA     IDETMP          ; save the value to write for a minute
        TAD     [IDEOUT]        ; set ports A and B to output mode
        PWCR                    ; write the PPI control register
        TAD     @IDEWRR         ; get the IDE register address
        ISZ     IDEWRR          ; (skip it when we return)
        PWPC                    ; send the address to the drive via port C

;   Note that we don'e bother to drive the upper data byte (D8..D15) with any
; particular value.  The PPI will have set these bits to zero when we changed
; the mode to output, but the drive will ignore them anyway.
        TAD     IDETMP          ; get the original data back
        PWPB                    ; (port B drives DD0..DD7)
        TAD     [SETDWR]        ; assert DIOW
        PWCR                    ; ...
        TAD     [CLRDWR]        ; and then clear it
        PWCR                    ; ...

;   We always leave our side of the PPI data bus (e.g. ports A and B) in
; input mode to avoid any accidental contention should the drive decide it
; wants to output data for some unknown reason.
        TAD     [IDEINP]        ; set ports A and B to input mode
```

```
        PWCR                        ;  ... (but C is still an output)
        JMP     @IDEWRR             ; that's it!
        .TITLE  Read IDE Register


;   This routine will read one IDE drive register and return the value in the
; AC.  all IDE registers, with the exception of the data register, are always
; eight bits wide so there's no need to worry about the upper byte here.  We
; simply ignore it.  The address of the register to be read should be passed
; inline, following the call to this procedure.
;
;CALL
;       JMS     IDERDR
;        xxxx           ; IDE register number, including CS1Fx and CS3Fx bits
;       <return here with 8 bit value in AC>
;
IDERDR: 0                           ; CALL HERE WITH A JMS!!
        CLA                         ; just in case...
        TAD     [IDEINP]            ; set ports A and B to input
        PWCR                        ; ... (this should be unnecessary,
        TAD     @IDERDR             ; get the IDE register address
        ISZ     IDERDR              ; (and don't forget to skip it!)
        PWPC                        ; send it to the drive via port C
        TAD     [SETDRD]            ; assert DIOR
        PWCR                        ; ...
        NOP                         ; give the drive and 8255 time to settle
        PRPB                        ; capture D0..D7
        AND     [377]               ; make sure we don't get noise in DX0..DX3
        DCA     IDETMP              ; and save that for a minute
        TAD     [CLRDRD]            ; now deassert DIOR
        PWCR                        ; ...
        TAD     IDETMP              ; get the data back
        JMP     @IDERDR             ; and return it in the AC

; Local storage for RD/IDEWRR...
IDETMP: .BLOCK  1                   ; a temporary for saving the AC

        .PAGE
        .TITLE  I/O Buffer Management


;   This routine is used parse the argument list for ROM calls that take OS/8
; handler like argument lists, primarily the RAM disk and IDE disk I/O calls.
;  It will do a GETARG and store the first argument, which contains the R/W
; bit, page count, buffer field and unit number, in MUUO.  It extracts the
; buffer field from this argument, builds a CDF instruction, and stores that
; at BUFCDF for later use.  It also extracts the page count from this
; argument, converts it to a negative number, and stores the result at RWCNT.
; Finally, it does another GETARG to fetch the address of the caller's buffer
; and stores that at BUFPTR.
SETBUF: .PUSHJ  @[GETARG]           ; get the first argument
        DCA     MUUO                ; save that - it's got lots of useful bits!
        TAD     MUUO                ; get the field bits from MUUO
        AND     [70]                ; ...
        TAD     [CDF 0]             ; make a CDF instruction out of them
        DCA     BUFCDF+1            ; and save them for later
        TAD     MUUO                ; get the page count from the call
        AND     [3700]              ; ...
        SNA                         ; is it zero ?
         NL4000                     ; yes - that means to transfer a full 32 pages
        BSW                         ; right justify the page count
        CIA                         ; make it negative for an ISZ
        DCA     RWCNT               ; ...
        .PUSHJ  @[GETARG]           ; get the buffer pointer from the argument list
        TAD     [-1]                ; correct for pre-incrementing auto-index
        DCA     BUFPTR              ; and save that
        DCA     BUFPNL              ; this buffer is always in main memory
        .POPJ                       ; all done for now


;   This routine will set up BUFPTR, BUFCDF, RWCNT and BUFPNL to point to
; our own internal buffer in panel memory at DSKBUF.  This is used by the
```

```
; disk dump, disk load, format and boot commands when they need to read or
; write disk blocks without distrubing main memory.
PNLBUF: TAD     [DSKBUF-1]      ; point to the disk buffer
        DCA     BUFPTR          ; set the buffer address for DISKRD/DISKWR
        TAD     [CDF 1]         ; this buffer lives in our field 1
        DCA     BUFCDF+1        ; ...
        NLM2                    ; the buffer size is always 2 pages
        DCA     RWCNT           ; ...
        NL7777                  ; write this data to PANEL memory!
        DCA     BUFPNL          ; ...
        .POPJ                   ; and we're done


;   This little routine is called, via a JMS instruction (not a .PUSHJ!) to
; change the DF to the field of the user's buffer.  In addition, if the
; BUFPNL flag is not set, it will execute a CPD instruction so that buffer
; data is stored in main memory.  This is the usual case.
BUFCDF: 0                       ; call here with a JMS
        NOP                     ; gets over written with a CDF instruction
        CLA                     ; just in case
        TAD     BUFPNL          ; is the panel buffer flag set?
        SNA CLA                 ; ???
         CPD                    ; no - address main memory now
        JMP     @BUFCDF         ; ...
        .TITLE  Copy Memory ROM Calls


;   This ROM function can copy up to 4096 words from any field in either main
; or panel memory to any other address and field in either main or panel
; memory.  It can be used to move data and/or code into panel memory and
; back again, or simply to move one part of main memory to another.
;
;CALL:
;       PR0
;        0010           / copy memory subfunction
;        p0n0           / source field and memory space
;        <address>      / source address
;        p0n0           / destination field and memory space
;        <address>      / destination address
;        <word count>   / number of words to be transferred
;
;   The source and destination field words each contain the field number in
; bits 6..8, and a flag in bit 0 which is one for panel memory and zero for
; main memory.  The last word of the argument list is the number of words
; to be copied - a value of zero copies 4096 words.

; Set up the source address...
MEMMOV: .PUSHJ  @[GETARG]       ; get the source field
        CLL                     ; make sure the link is in a known state
        TAD     [4000]          ; put the panel/main memory flag in the LINK
        AND     [70]            ; make a CDF instruction
        TAD     [CDF 0]         ; ...
        DCA     SRCCDF          ; ...
        TAD     [CPD]           ; assume the source is in main memory
        SZL                     ; but is it really ?
         TAD     [SPD-CPD]      ; no - use panel memory
        DCA     SRCSPD          ; ...
        .PUSHJ  @[GETARG]       ; get the buffer address
        TAD     [-1]            ; correct for pre-increment auto index
        DCA     XX1             ; ...

; Set up the destination address...
        .PUSHJ  @[GETARG]       ; get the destination field
        CLL                     ; make sure the link is in a known state
        TAD     [4000]          ; put the panel/main memory flag in the LINK
        AND     [70]            ; make a CDF instruction
        TAD     [CDF 0]         ; ...
        DCA     DSTCDF          ; ...
        TAD     [CPD]           ; assume the destination is in main memory
        SZL                     ; but is it really ?
         TAD     [SPD-CPD]      ; no - use panel memory
        DCA     DSTSPD          ; ...
```

```
        .PUSHJ  @[GETARG]       ; get the buffer address
        TAD     [-1]            ; correct for pre-increment auto index
        DCA     XX2             ; ...

; And finally the word count...
        .PUSHJ  @[GETARG]       ; ...
        CIA                     ; make it negative for ISZ
        DCA     XFRCNT          ; ...

; This loop does the actual work of copying data!
SRCCDF: NOP                     ; over written with a CDF instruction
SRCSPD: NOP                     ; over written with a SPD/CPD IOT
        TAD     @XX1            ; get a word of source data
DSTCDF: NOP                     ; over written with a CDF instruction
DSTSPD: NOP                     ; overwritten with a SPD/CPD IOT
        DCA     @XX2            ; and store the word
        ISZ     XFRCNT          ; have we done them all ?
         JMP     SRCCDF         ; no - keep copying

; All done!
        SPD                     ; be sure the field and memory space are safe
        CDF     1               ; ...
        CLL CLA                 ; and always return success
        .POPJ                   ; ...

        .PAGE
        .TITLE  Free Space for Future Expansion!

        .PAGE   16
        .TITLE  Command Names Table


;   This table gives the names of all the commands known to the monitor.  Each
; entry consists of a one or two letter command name, in SIXBIT, followed by
; the address of a routine to execute it.  Although this table is stored in
; field 1, all the command routines are implicitly in field zero!  The zero
; entry at the end is a "catch all" that is called if none of the previous
; names match, and points to an error routine.  With the exception of this last
; entry, the order of the table is not significant.
CMDTBL:
        .SIXBIT /H /            ; Help
        HELP                    ; ...
        .SIXBIT /RP/            ; RePeat
        REPEAT                  ; ...
        .SIXBIT /E /            ; Examine
        EMEM                    ; ...
        .SIXBIT /EP/            ; Examine Panel memory
        EPMEM                   ; ...
        .SIXBIT /D /            ; Deposit
        DMEM                    ; ...
        .SIXBIT /DP/            ; Deposit Panel memory
        DPMEM                   ; ...
        .SIXBIT /ER/            ; Examine Register
        EREG                    ; ...
        .SIXBIT /DR/            ; Deposit Register
        DREG                    ; ...
        .SIXBIT /BM/            ; Block Move
        BMOVE                   ; ...
        .SIXBIT /CK/            ; ChecKsum
        CKMEM                   ; ...
        .SIXBIT /WS/            ; Word Search
        SEARCH                  ; ...
        .SIXBIT /CM/            ; Clear Memory
        CMEM                    ; ...
        .SIXBIT /FM/            ; Fill Memory
        FLMEM                   ; ...
        .SIXBIT /BL/            ; Breakpoint List
        BLIST                   ; ...
        .SIXBIT /BP/            ; BreakPoint
        BPTCOM                  ; ...
        .SIXBIT /BR/            ; Breakpoint Remove
        BREMOV                  ; ...
```

```
        .SIXBIT /C /            ; Continue
        CONTCM                  ; ...
        .SIXBIT /SI/            ; Single Instruction with no trace
        SNCOM                   ; ...
        .SIXBIT /ST/            ; STart
        START                   ; ...
        .SIXBIT /P /            ; Proceed
        PROCEE                  ; ...
        .SIXBIT /TR/            ; single instruction with TRace
        SICOM                   ; ...
        .SIXBIT /VE/            ; VErsion (of monitor)
        VECOM                   ; ...
        .SIXBIT /TW/            ; Terminal Width
        TWCOM                   ; ...
        .SIXBIT /TP/            ; Terminal Page
        TPCOM                   ; ...
        .SIXBIT /EX/            ; EXECUTE (IOT instruction)
        XCTCOM                  ; ...
        .SIXBIT /MR/            ; MASTER RESET
        CLRCOM                  ; ...
        .SIXBIT /LP/            ; LOAD PAPER (tape from console)
        CONLOD                  ; ...
        .SIXBIT /DD/            ; Disk (IDE) Dump
        DDDUMP                  ; ...
        .SIXBIT /RD/            ; Disk (RAM) Dump
        RDDUMP                  ; ...
        .SIXBIT /DL/            ; Disk (IDE) Load
        DLLOAD                  ; ...
        .SIXBIT /RL/            ; Disk (RAM) Load
        RLLOAD                  ; ...
        .SIXBIT /DF/            ; Disk (IDE) Format
        DFRMAT                  ; ...
        .SIXBIT /RF/            ; Disk (RAM) Format
        RFRMAT
        .SIXBIT /B /            ; Bootstrap ram disk
        BOOT
        .SIXBIT /PM/            ; Partition Map
        PMEDIT                  ; ...
        .SIXBIT /PC/            ; Partition Copy
        PCOPY                   ; ...
        0000                    ; This must always be the last entry
        COMERR                  ; Where to go if none of the above matches
        .TITLE  Argument Tables for Various Commands


;   This table gives a list of the legal register names for the ER (Examine
; Register) command...
ENAMES: .SIXBIT /AC/            ; The AC
        TYPEAC
        .SIXBIT /PC/            ; The PC
        TYPEPC
        .SIXBIT /MQ/            ; The MQ
        TYPEMQ
        .SIXBIT /PS/            ; The processor status
        TYPEPS
        .SIXBIT /SR/            ; The switch register
        TYPESR
        0000                    ; None of the above
        COMERR

;   This table gives a list of the legal register names for the DR (deposit
; register) command...
DNAMES: .SIXBIT /AC/            ; The AC
        DAC
        .SIXBIT /PC/            ; The PC
        DPC
        .SIXBIT /MQ/            ; The MQ
        DMQ
        .SIXBIT /PS/            ; The flags
        DPS
        .SIXBIT /SR/            ; The switch register
        DSR
```

```
        0000                    ; None of the above
        COMERR

; This table is a list of the arguments to the B (BOOT) command...
BNAMES: .SIXBIT /VM/            ; VMA0
        BTVMA0
        .SIXBIT /ID/            ; IDA0
        BTIDA0
        0000                    ; end of list
        COMERR
        .TITLE  Messages

; General purpose messages...
CKSMSG: .TEXT   /Checksum = /
MEMMSG: .TEXT   /?Memory error at /
ERRILV: .TEXT   /Illegal value/
ERRSRF: .TEXT   /Search fails/
ERRRAN: .TEXT   /Wrong order/
ERRWRP: .TEXT   /Wrap around/
SKPMSG: .TEXT   /Skip /
ERRDIO: .TEXT   \?I/O Error \
ERRCKS: .TEXT   /Checksum error/
ERRNBT: .TEXT   /No bootstrap/
ERRNDK: .TEXT   /No disk/

; Program trap messages...
BPTMSG: .TEXT   /%Breakpoint at /
PR0MSG: .TEXT   /?Illegal PR0 function at /
BRKMSG: .TEXT   /%Break at /
PRNMSG: .TEXT   /?Panel trap at /
HLTMSG: .TEXT   /?Halted at /
TRPMSG: .TEXT   /?Unknown trap at /

; Breakpoint messages...
ERRNBP: .TEXT   /None set/
ERRNST: .TEXT   /Not set/
ERRAST: .TEXT   /Already set/
ERRBTF: .TEXT   /Table full/

; Register names...
ACNAME: .TEXT   /AC>/
PCNAME: .TEXT   /PC>/
MQNAME: .TEXT   /MQ>/
IRNAME: .TEXT   /IR>/
SRNAME: .TEXT   /SR>/
PSNAME: .TEXT   /PS>/
SP1NAM: .TEXT   /SP1>/
SP2NAM: .TEXT   /SP2>/

; Disk formatting status messages...
FCFMSG: .TEXT   \Format unit/partition \
FM1MSG: .TEXT   /Writing /
FM2MSG: .TEXT   / Verifying /
FM3MSG: .TEXT   / Done/
ERRDSK: .TEXT   \?Verification error, block/page \

; Partition copy messages....
CCFMSG: .TEXT   \Overwrite partition \
CP1MSG: .TEXT   /Copying /
CP2MSG=FM3MSG

; Partition map messages...
PM1MSG: .TEXT   /Unit /
PM2MSG: .TEXT   / -> Partition /

; Device names that get printed by the boot sniffer...
VMAMSG: .TEXT   /-VMA0/
IDAMSG: .TEXT   /-IDA0/

; System name message...
SYSNM1: .TEXT   /SBC6120 ROM Monitor V/
SYSNM2: .TEXT   / Checksum /
```

```
SYSNM3: .TEXT   / \d \h/
SYSCRN: .TEXT   /Copyright (C) 1983-2003 Spare Time Gizmos.  All rights reserved./

; RAM disk status message...
RAMMS1: .TEXT   /NVR: /
RAMMS3: .TEXT   /KB - Battery /
BOKMSG: .TEXT   /OK/
BFAMSG: .TEXT   /FAIL/

; IDE disk status message...
IDEMS1: .TEXT   /IDE: /
IDEMS2: .TEXT   /MB - /
IDEMS3: .TEXT   /Not detected/
IDEMS4: .TEXT   /Not supported/
        .TITLE  Help Text


;   This table is used by the HELP command to generate a page of text
; describing the monitor commands.  Each word is a pointer to a text string,
; also in field 1, which contains a single line of text, usually a description
; of one command.  The table ends with a zero word.
HLPLST:
        ; Examine/Deposit commands...
        .DATA   HLPEDC
        .DATA   HLPE, HLPEP, HLPER, HLPD, HLPDP, HLPDR
        ; Memory commands...
        .DATA   HLPNUL, HLPMEM
        .DATA   HLPBM, HLPCK, HLPWS, HLPFM, HLPCM
        ; Breakpoint commands...
        .DATA   HLPNUL, HLPBPC
        .DATA   HLPBP, HLPBR, HLPBL, HLPP
        ; Program control commands...
        .DATA   HLPNUL, HLPPCC
        .DATA   HLPST, HLPC, HLPSI, HLPTR, HLPEX, HLPMR
        ; Disk commands...
        .DATA   HLPNUL, HLPDSK
        .DATA   HLPLP, HLPRD, HLPRL, HLPRF, HLPDD, HLPDL, HLPDF
        .DATA   HLPPC, HLPPM, HLPB
        ; Other (miscellaneous) commands...
        .DATA   HLPNUL, HLPMSC
        .DATA   HLPTW, HLPTP, HLPVE, HLPSEM, HLPRP, HLPDOL
        ; Special control characters...
        .DATA   HLPNUL, HLPCTL
        .DATA   HLPCTS, HLPCTQ, HLPCTO, HLPCTC, HLPCTH, HLPRUB, HLPCTR, HLPCTU
HLPNUL: .DATA   0

; Examine/Deposit commands...
HLPEDC: .TEXT   /EXAMINE AND DEPOSIT COMMANDS/
HLPE:   .TEXT   /E  aaaaa[-bbbbb] [, ccccc]\t-> Examine main memory/
HLPEP:  .TEXT   /EP aaaaa[-bbbbb] [, ccccc]\t-> Examine panel memory/
HLPER:  .TEXT   /ER [rr]\t\t\t\t-> Examine register/
HLPD:   .TEXT   /D  aaaaa bbbb, [cccc, ...]\t-> Deposit in main memory/
HLPDP:  .TEXT   /DP aaaaa bbbb, [cccc, ...]\t-> Deposit in panel memory/
HLPDR:  .TEXT   /DR xx yyyy\t\t\t-> Deposit in register/

; Memory commands...
HLPMEM: .TEXT   /MEMORY COMMANDS/
HLPBM:  .TEXT   /BM aaaaa-bbbbb ddddd\t\t-> Move memory block/
HLPCK:  .TEXT   /CK aaaaa-bbbbb\t\t\t-> Checksum memory block/
HLPWS:  .TEXT   /WS vvvv [aaaaa-bbbbb [mmmm]]\t-> Search memory/
HLPFM:  .TEXT   /FM vvvv [aaaaa-bbbbb]\t\t-> Fill memory/
HLPCM:  .TEXT   /CM [aaaaa-bbbbb]\t\t-> Clear memory/

; Breakpoint commands...
HLPBPC: .TEXT   /BREAKPOINT COMMANDS/
HLPBP:  .TEXT   /BP aaaaa\t\t\t-> Set breakpoint/
HLPBR:  .TEXT   /BR [aaaaa]\t\t\t-> Remove breakpoint/
HLPBL:  .TEXT   /BL\t\t\t\t-> List breakpoints/
HLPP:   .TEXT   /P\t\t\t\t-> Proceed past breakpoint/

; Program control commands...
HLPPCC: .TEXT   /PROGRAM CONTROL COMMANDS/
```

```
HLPST:  .TEXT   /ST [aaaaa]\t\t\t-> Start main memory program/
HLPC:   .TEXT   /C\t\t\t\t-> Continue execution/
HLPSI:  .TEXT   /SI\t\t\t\t-> Single instruction/
HLPTR:  .TEXT   /TR\t\t\t\t-> Trace one instruction/
HLPEX:  .TEXT   /EX 6xxx\t\t\t\t-> Execute an IOT instruction/
HLPMR:  .TEXT   /MR\t\t\t\t-> Master reset/

; Disk commands...
HLPDSK: .TEXT   /DISK COMMANDS/
HLPLP:  .TEXT   /LB\t\t\t\t-> Load a BIN paper tape/
HLPRD:  .TEXT   /RD u [pppp [cccc]]\t\t-> Dump RAM disk page/
HLPRL:  .TEXT   /RL u\t\t\t\t-> Download RAM disk/
HLPRF:  .TEXT   /RF u\t\t\t\t-> Format RAM disk/
HLPDD:  .TEXT   /DD pppp [bbbb [cccc]]\t\t-> Dump IDE disk block/
HLPDL:  .TEXT   /DL pppp\t\t\t\t-> Download IDE disk/
HLPDF:  .TEXT   /DF pppp\t\t\t\t-> Format IDE disk/
HLPPM:  .TEXT   /PM [u] [pppp]\t\t\t-> Edit or review IDE partition map/
HLPPC:  .TEXT   /PC ssss dddd\t\t\t-> Copy partition ssss to dddd/
HLPB:   .TEXT   /B [dd]\t\t\t\t-> Boot RAM or IDE disk /

; Other (miscellaneous) commands...
HLPMSC: .TEXT   /MISCELLANEOUS COMMANDS/
HLPTW:  .TEXT   /TW nn\t\t\t\t-> Set the console width/
HLPTP:  .TEXT   /TP nn\t\t\t\t-> Set the console page length/
HLPVE:  .TEXT   /VE\t\t\t\t-> Show firmware version/
HLPSEM: .TEXT   /aa; bb; cc; dd ...\t\t-> Combine multiple commands/
HLPRP:  .TEXT   /RP [nn]; A; B; C; ...\t\t-> Repeat commands A, B, C/
HLPDOL: .TEXT   /!any text...\t\t\t-> Comment text/

; Special control characters...
HLPCTL: .TEXT   /SPECIAL CHARACTERS/
HLPCTS: .TEXT   /Control-S (XOFF)\t\t-> Suspend terminal output/
HLPCTQ: .TEXT   /Control-Q (XON)\t\t\t-> Resume terminal output/
HLPCTO: .TEXT   /Control-O\t\t\t-> Suppress terminal output/
HLPCTC: .TEXT   /Control-C\t\t\t-> Abort current operation/
HLPCTH: .TEXT   /Control-H (Backspace)\t\t-> Delete the last character entered/
HLPRUB: .TEXT   /RUBOUT (Delete)\t\t\t-> Delete the last character entered/
HLPCTR: .TEXT   /Control-R\t\t\t-> Retype the current line/
HLPCTU: .TEXT   /Control-U\t\t\t-> Erase current line/
        .TITLE  Temporary Disk Buffer


;   The last two pages of field 1, addresses 17400 thru 17777, are used as a
; temporary disk buffer by the disk load, disk dump, disk format and boot
; commands.
        .PAGE   36
DSKBUF: .BLOCK  128.
        .PAGE   37
        .BLOCK  128.

        .END
```